

ALU LIBRARY

HP BASIC 6.2 Programming Guide



HP Part No. 98616-90010
Printed in USA

Notice

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

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

Warranty Information

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

Restricted Rights Legend

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

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

Copyright © Hewlett-Packard Company 1987, 1988, 1989, 1990, 1991

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Copyright © AT&T Technologies, Inc. 1980, 1984, 1986

Copyright © The Regents of the University of California 1979, 1980, 1983,
1985-86

This software and documentation is based in part on the Fourth Berkeley
Software Distribution under license from the Regents of the University of
California.

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

Printing History

First Edition - June 1991

Contents

1. Introduction	
Some Important Notations	1-2
What's In This Manual?	1-2
2. Program Structure and Flow	
The Program Counter	2-2
Sequence	2-3
Linear Flow	2-3
Halting Program Execution	2-3
Simple Branching	2-6
Using GOTO	2-6
Using GOSUB	2-7
Line Labels and Comments	2-8
Selection	2-9
Conditional Execution of One Segment	2-10
Prohibited Statements	2-11
Multiple-Line Conditional Segments	2-11
Choosing One of Two Segments	2-12
Choosing One of Many Segments	2-13
Using SELECT	2-13
Repetition	2-15
Fixed Number of Iterations	2-15
Conditional Number of Iterations	2-17
Arbitrary Exit Points	2-19
WRONG:	2-21
RIGHT:	2-21
Event-Initiated Branching	2-21
Types of Events	2-22
Example of Event-Initiated Branching	2-24
Example of Using the Knob	2-26

Deactivating Events	2-27
Disabling Events	2-29
Chaining Programs	2-30
Using LOAD	2-30
Using GET	2-30
Example of Chaining with GET	2-31
Program-to-Program Communications	2-32
3. Numeric Computation	
Numeric Data Types	3-1
REAL Data Type	3-1
INTEGER Data Type	3-2
COMPLEX Data Type	3-2
Numeric Variables	3-2
Naming Variables	3-3
Declaring Variables	3-3
Assigning Variables	3-4
Implicit Type Conversions	3-4
Precision and Accuracy: The Machine Limits	3-7
Internal Numeric Formats	3-8
Evaluating Scalar Expressions	3-9
The Hierarchy	3-9
The Delayed Assignment Surprise	3-11
Operators	3-11
Expressions as Pass Parameters	3-12
Comparisons Between Two REAL or COMPLEX Values	3-12
Numerical Functions	3-13
Arithmetic Functions	3-13
Array Functions	3-14
Exponential Functions	3-14
Trigonometric Functions	3-14
Trigonometric Modes: Degrees and Radians	3-15
Hyperbolic Functions	3-15
Binary Functions	3-16
Limit Functions	3-16
Rounding Functions	3-16
Rounding Errors Resulting from Comparisons	3-17
Random Number Function	3-18

Complex Functions	3-19
Creating COMPLEX Values	3-19
Evaluating COMPLEX Numbers	3-19
COMPLEX Arguments and the Trigonometric Mode	3-20
Determining the Parts of COMPLEX Numbers	3-20
Converting from Rectangular to Polar Coordinates	3-21
Time and Date Functions	3-22
Base Conversion Functions	3-23
General Functions	3-24
Floating-Point Math	3-25

4. Numeric Arrays

Dimensioning an Array	4-2
Some Examples of Arrays	4-3
Problems with Implicit Dimensioning	4-6
Finding Out the Dimensions of an Array	4-6
Using Individual Array Elements	4-7
Assigning an Individual Array Element	4-7
Extracting Single Values From Arrays	4-7
Filling Arrays	4-8
Assigning Every Element in an Array the Same Value	4-8
Using the READ Statement to Fill an Entire Array	4-8
Copying Entire Arrays into Other Arrays	4-9
Printing Arrays	4-11
Printing an Entire Array	4-11
Examples of Formatting Arrays for Display	4-11
Passing Entire Arrays	4-13
Copying Subarrays	4-13
Subarray Specifier	4-14
Copying an Array into a Subarray	4-16
Copying a Subarray into an Array	4-17
Copying a Subarray into another Subarray	4-18
Copying a Portion of an Array into Itself	4-19
Rules for Copying Subarrays	4-20
Redimensioning Arrays	4-21

5. String Manipulation	
Details About Strings	5-1
String Storage	5-2
String Arrays	5-3
Evaluating Expressions Containing Strings	5-3
Evaluation Hierarchy	5-3
String Concatenation	5-4
Relational Operations	5-4
Substrings	5-5
Single-Subscript Substrings	5-5
Double-Subscript Substrings	5-6
Special Considerations	5-7
String-Related Functions	5-9
Special Note for Localized BASIC	5-9
Current String Length	5-9
Maximum String Length	5-10
Substring Position	5-10
String-to-Numeric Conversion	5-11
Numeric-to-String Conversion	5-12
CRT Character Set	5-13
String Functions	5-14
String Reverse	5-14
String Repeat	5-15
Trimming a String	5-15
Case Conversion	5-16
Copying String Arrays and Subarrays	5-16
Searching and Sorting	5-17
Sorting by Substrings	5-18
Adding Items to a Sorted List	5-19
Sorting by Multiple Keys	5-20
Sorting to a Vector	5-21
Reordering an Array	5-21
Searching for Strings	5-22
Searching String Arrays	5-22
Number-Base Conversion	5-23
Introduction to Lexical Order	5-24
Why Lexical Order?	5-24
How It Works	5-24

The ASCII Character Set	5-25
Displaying Control Characters	5-25
The Extended Character Set	5-26
Display Enhancement Characters	5-26
Alternate CRT Characters	5-27
Finding “Missing” Characters	5-28
Predefined Lexical Order	5-28
6. Subprograms and User-Defined Functions	
An Example	6-1
A Closer Look at Subprograms	6-3
Calling and Executing a Subprogram	6-3
Differences Between Subprograms and Subroutines	6-4
Subprogram Location	6-4
Subprogram and User-Defined Function Names	6-5
Difference Between a User-Defined Function and a Subprogram	6-5
Program/Subprogram Communication	6-6
Parameter Lists	6-6
Formal Parameter Lists	6-6
Pass Parameter Lists	6-7
Passing By Value vs. Passing By Reference	6-7
Example Pass and Corresponding Formal Parameter Lists .	6-8
OPTIONAL Parameters	6-9
COM Blocks	6-11
COM vs. Pass Parameters	6-12
Hints for Using COM Blocks	6-13
Calling Subprograms Using a String Name	6-15
Context Switching	6-16
Variable Initialization	6-17
Subprograms and Softkeys	6-17
Subprograms and the RECOVER Statement	6-18
Calling Subprograms from the Keyboard	6-20
Using Subprogram Libraries	6-20
Why Use Subprogram Libraries?	6-20
Listing the Subprograms in a PROG File	6-20
Loading Subprograms	6-21
Loading Subprograms One at a Time	6-21
Loading Several Subprograms at Once	6-22

Loading Subprograms Prior to Execution	6-22
Deleting Subprograms	6-23
Editing Subprograms	6-25
Inserting Subprograms	6-25
Deleting Subprograms	6-26
Merging Subprograms	6-26
SUBEND and FNEND	6-27
Recursion	6-27
Top-Down Design	6-28

7. Data Storage and Retrieval

Storing Data in Programs	7-1
Storing Data in Variables	7-2
Data Input by the User	7-2
Using DATA and READ Statements	7-2
Examples	7-4
Storage and Retrieval of Arrays	7-4
Moving the Data Pointer	7-5
File Input and Output (I/O)	7-6
Brief Comparison of Available File Types	7-7
Overview of File I/O	7-9
A Closer Look at General File Access	7-11
Opening an I/O Path	7-12
Assigning Attributes	7-13
Closing I/O Paths	7-15
Locking Files	7-15
Example Locking and Unlocking of an SRM File	7-16
BASIC/UX Specifics on Locking SRM Files	7-16
Locking HFS Files (BASIC/UX ONLY)	7-17
Extended Access of Directories	7-18
Cataloging to a String Array	7-19
Getting an "Extended" Catalog of a LIF or HFS Disk	7-20
Getting a Count of Files Cataloged	7-21
Suppressing the Catalog Header	7-21
Cataloging Selected Files	7-22
CAT with the SELECT Secondary	7-22
CAT with Wildcards	7-23
Getting a Count of Selected Files	7-24

Skipping Selected Files	7-24
HFS File Buffering (BASIC/UX only)	7-26
What Is File Buffering?	7-26
Turning File Buffering On and Off	7-27
Flushing the File Buffer	7-27
DFS File Buffering (BASIC/DOS only)	7-27

8. Graphics Techniques

Why Graphics?	8-2
Drawing Lines	8-3
Scaling	8-5
Defining a Viewport	8-9
GDUs and UDUs	8-9
Specifying the Viewport	8-10
Labeling a Plot	8-11
Axes and Tick Marks	8-13
Using Graphics Effectively	8-16
More on Defining a Viewport	8-16
Special Considerations about Scaling	8-18
Why Does This Math Overflow Error Occur?	8-19
More on Labeling a Plot	8-19
Axes and Grids	8-30
Strategy: Axes Versus Grids	8-33
Miscellaneous Graphics Concepts	8-38
Clipping	8-38
Drawing Modes	8-39
Selecting Line Types	8-41
Storing and Retrieving Images	8-43
Data-Driven Plotting	8-45
Translating and Rotating a Drawing	8-49
Incremental Plotting	8-53
Drawing Polygons	8-54
But I Don't Want Polygon Closure	8-55
Rectangles	8-55
User-Defined Characters	8-56
Multi-Plane Bit-Mapped Displays	8-59
The Graphics Write-Enable Mask	8-59
The Graphics Display-Enable Mask	8-60

The Alpha Masks	8-60
Interactions Between Alpha and Graphics Masks	8-61
Disabling and Enabling Alpha Scrolling	8-64
Introduction to Color Graphics	8-65
Non-Color Mapped Color	8-65
Color Mapped Color	8-66
Default Colors	8-67
Changing Default Colors	8-68
Fill Colors	8-69
9. Using Printers and Plotters for Generating Graphics	
Dumping Raster Images to Printers	9-2
Dumping to HP Raster Interface Standard Devices	9-2
Dumping from a Color Display	9-3
Dumping from a High-Resolution Display	9-3
Using the DUMP GRAPHICS Key	9-3
Aborting Graphics Dumps	9-3
Expanded Dumps	9-4
Dumping Displays with Non-Square Pixels	9-4
Dumping to Non-Standard Printers	9-5
Example of Dumping to a Non-standard Printer	9-5
Negative Images	9-6
Dumping Raster Images to a File	9-6
Specifying the File that Receives the Raster Dump	9-6
Example	9-7
Using Plotters	9-7
Selecting a Plotter	9-8
Plotter Graphics with HPGL Commands	9-8
Example: Controlling Pen Speed	9-8
Example: Controlling Pen Force	9-9
Example: Selecting Character Sets	9-9
Error Detection when Using HPGL Commands	9-9
Plotting to Files	9-10
Plotter Paper Sizes	9-10
Limitations	9-11
Using GSEND with PLOTTER IS Files	9-11
Using SRM Plotter Spoolers	9-12
What Are Spoolers?	9-12

Setting Up a Plotter Spooler	9-12
Preparing Plotters	9-12
Plotter Spooling	9-12
Example of Plotting to a File	9-13
Checking the Spooler's Status	9-13
Aborting Plotting In Progress	9-14
Dumping Graphics to a Printer Spooler	9-14
Using a Plotter with BASIC/UX	9-14
Prerequisites	9-15
Assigning a Plotter Spooler as the Output Device	9-15
Sending Graphics Output to the Plotter Spooler	9-16
Assigning a Window as the Plotting Device	9-16
Sending a Graphics Plot to a BASIC/UX Window	9-17

10. Using a Printer

Installing, Configuring, and Verifying Your Printer	10-1
Selecting the System Printer	10-1
What Are Device Selectors?	10-3
Using Device Selectors to Select Printers	10-3
Using Control Characters and Escape Sequences	10-4
Control Characters	10-4
Escape-Code Sequences	10-5
Formatted Printing	10-5
Using Images	10-7
Numeric Image Specifiers	10-8
String Image Specifiers	10-10
Additional Image Specifiers	10-11
Special Considerations	10-11
Using Printers through the SRM Spooler	10-12
Using an SRM Spooler	10-12
SRM Printer Spooling Using PRINTER IS	10-13
Writing Files to SRM Spooler Directories	10-13
Sending Program Output to a Shared SRM Printer	10-13
Appearance of Output	10-14
Using an HP-UX Printer Spooler (BASIC/UX Only)	10-15
Prerequisites	10-15
A Simple Example	10-15

11. Using the Clock and Timers	
BASIC CLOCK Binary	11-1
Do You Have a Battery-Powered Clock?	11-1
Initial Clock Value	11-2
Clock Range and Accuracy	11-2
Reading the Clock	11-3
Determining the Date and Time of Day	11-3
Setting the Time, Date, and Time Zone	11-4
BASIC/WS and HP-UX Clock Compatibility	11-4
Setting the Time and Date	11-4
The BASIC/UX “Local” Clock	11-4
Clock Time Format	11-4
Setting Only the Time	11-6
Setting Only the Date	11-8
Using Clock Functions and Example Programs	11-11
Day of the Week	11-13
Days Between Two Dates	11-13
Interval Timing	11-13
Branching on Clock Events	11-14
Clock Resolution	11-15
Cycles and Delays	11-15
Time of Day	11-16
Priority Restrictions	11-17
Branching Restrictions	11-18
12. Handling Errors	
Overview of Error Responses	12-1
Anticipating Operator Errors	12-2
Boundary Conditions	12-2
REAL and COMPLEX Numbers and Comparisons	12-4
Trapping Errors with BASIC Programs	12-5
Setting Up Error Service Routines	12-5
(ON/OFF ERROR)	12-5
Choosing a Branch Type	12-5
Scope of Error Trapping and Recovery	12-6
ON ERROR Execution at Run-Time	12-6
ON ERROR Priority	12-6
Disabling Error Trapping (OFF ERROR)	12-6

Determining Error Number and Location (ERRN, ERRLN, ERRL, ERRDS, ERRM\$)	12-7
A Closer Look at ON ERROR GOSUB	12-8
A Closer Look At ON ERROR GOTO	12-9
A Closer Look At ON ERROR CALL	12-11
Cannot Pass Parameters Using ON ERROR CALL	12-12
Using ERRLN and ERRL in Subprograms	12-12
A Closer Look At ON ERROR RECOVER	12-13
Simulating Errors (CAUSE ERROR)	12-14
Example of Simulating an Error	12-15
CAUSE ERROR and Error Numbers 1001 thru 1080	12-15
Clearing Error Conditions (CLEAR ERROR)	12-16
Error handling for File Name Expansion	12-17
Recording Interaction and Troubleshooting Messages	12-18

13. Debugging Programs

Using Live Keyboard	13-2
Executing Commands While a Program Is Running	13-2
Using Program Variables	13-2
Calling Subprograms	13-4
Pausing and Continuing a Program	13-5
Keyboard Commands Disallowed During Program Execution	13-5
Cross References	13-6
Generating a Cross-Reference Listing	13-6
Example Program and Cross Reference	13-6
Unused Entries	13-7
Single-Stepping a Program	13-10
Tracing	13-11
TRACE ALL	13-11
PRINTALL IS	13-14
TRACE PAUSE	13-14
TRACE OFF	13-15
The CLR I/O (Break) Key	13-15



14. Introduction to I/O	
Interfacing Concepts	14-1
Terminology	14-1
Why Do You Need an Interface?	14-3
Electrical and Mechanical Compatibility	14-4
Data Compatibility	14-5
Timing Compatibility	14-5
Additional Interface Functions	14-5
Data Representations	14-6
Bits and Bytes	14-6
Representing Numbers	14-7
Representing Characters	14-8
Representing Signed Integers	14-8
Internal Representation of Integers	14-9
ASCII Representation of Integers	14-10
Representing Real Numbers	14-11
Internal Representation of Real Numbers	14-11
ASCII Representation of Real Numbers	14-12
The I/O Process	14-12
I/O Statements and Parameters	14-12
Specifying a Resource	14-12
Firmware	14-13
Registers	14-13
Data Handshake	14-14
I/O Examples	14-14
Example Output Statement	14-14
Source-Item Evaluation	14-15
Copying Data to the Destination	14-15
Example Enter Statement	14-16
Destination-Item Evaluation	14-17
Copying Data into the Destinations	14-17
Sharing Resources in a Multi-Tasking/Multi-User Environment (BASIC/UX only)	14-17
What Is Multi-Tasking?	14-18
What Does Multi-User Mean?	14-18
Using I/O Resources in a Multi-Tasking/Multi-User Environment	14-18
Directing Data Flow	14-19

Specifying a Resource	14-20
String-Variable Names	14-20
Formatted String I/O	14-21
Device Selectors	14-21
Select Codes of Built-In Interfaces (Series 200/300)	14-22
Select Codes of Optional Interfaces (Series 200/300)	14-23
HP-IB Device Selectors	14-24
Multiplexer Device Selectors (BASIC/UX Only)	14-24
I/O Path Names	14-25
I/O Path Variable Contents	14-27
Re-Assigning I/O Path Names	14-28
Closing I/O Path Names	14-28
I/O Path Names in Subprograms	14-29
Benefits of Using I/O Path Names	14-29
Execution Speed	14-29
Re-Directing Data	14-30
Attribute Control	14-31

15. Outputting Data

Introduction	15-1
Free-Field Outputs	15-1
The Free-Field Convention	15-2
Standard Numeric Format	15-2
Standard String Format	15-3
Item Separators and Terminators	15-3
Resultant Output	15-6
Resultant Output	15-7
Changing the EOL Sequence (Requires IO)	15-7
Using END in Freefield OUTPUT	15-9
Additional Definition	15-10
END with HP-IB Interfaces	15-10
END with the Data Communications Interface	15-11
Outputs that Use Images	15-12
The OUTPUT USING Statement	15-12
Images	15-13
Example of Using an Image	15-13
Image Definitions During Outputs	15-15
Numeric Images	15-15

Numeric Examples	15-17
String Images	15-21
String Examples	15-21
Binary Images	15-23
Binary Examples	15-24
Special-Character Images	15-25
Special-Character Examples	15-25
Termination Images	15-27
Termination Examples	15-27
Additional Image Features	15-28
Repeat Factors	15-29
Image Re-Use	15-30
Resultant Display	15-31
Nested Images	15-31
Resultant Output	15-32
END with OUTPUTs that Use Images	15-32
Additional END Definition	15-33
END with HP-IB Interfaces	15-33
END with Data Communications Interfaces	15-35
16. Entering Data	
Free-Field Enters	16-1
Item Separators	16-2
Item Terminators	16-2
Entering Numeric Data with the Number Builder	16-3
Entering String Data	16-8
Terminating Free-Field ENTER Statements	16-11
EOI Termination	16-12
Enters that Use Images	16-14
The ENTER USING Statement	16-14
Images	16-15
Example of an Enter Using an Image	16-15
Image Definitions During Enter	16-17
Numeric Images	16-17
String Images	16-19
Ignoring Characters	16-20
Binary Images	16-21
Terminating Enters that Use Images	16-22

Default Termination Conditions	16-23
EOI Re-Definition	16-23
Statement-Termination Modifiers	16-24
Additional Image Features	16-26
Repeat Factors	16-26
Image Re-Use	16-27
Nested Images	16-27
17. Registers	
Interface Registers	17-2
The STATUS Statement	17-2
The CONTROL Statement	17-3
I/O Path Registers	17-4
Summary of I/O Path Registers	17-8
For All I/O Path Names	17-8
I/O Path Names Assigned to a Device	17-8
I/O Path Names Assigned to an ASCII File	17-8
I/O Path Names Assigned to a BDAT File	17-9
I/O Path Names Assigned to an HP-UX File	17-9
I/O Path Names Assigned to a DOS File (BASIC/DOS Only)	17-10
I/O Path Names Assigned to a Buffer	17-11
I/O Path Names Assigned to HP-UX Special Files (BASIC/UX only)	17-12
Direct Interface Access	17-12
18. Interrupts and Timeouts	
Overview of Event-Initiated Branching	18-1
Types of Events	18-1
A Simple Example	18-2
Conditions Required for Initiating a Branch	18-5
Logging and Servicing Events	18-6
Software Priority	18-7
Changing System Priority	18-8
Hardware Priority	18-11
Servicing Pending Events	18-13
Interface Interrupts	18-15
Enabling Interrupt Events	18-16
Enabling and Disabling Events with WRITEIO	18-18

Service Requests	18-19
Interrupt Conditions	18-20
Interface Timeouts	18-21
Setting Up Timeout Events	18-22
Timeout Limitations	18-22
Timer Resolutions	18-22
No Default Timeout Parameter	18-23
Trapping HP-UX Signals from BASIC/UX	18-23
Relevant BASIC/UX Keywords	18-23

19. I/O Path Attributes

The FORMAT Attributes	19-1
Two FORMAT Attributes Are Available	19-2
FORMAT ON	19-2
FORMAT OFF	19-3
Assigning Default FORMAT Attributes	19-4
Specifying I/O Path Attributes	19-4
Restoring the Default Attributes	19-5
Additional Attributes	19-5
The APPEND Attribute	19-6
The BYTE and WORD Attributes	19-6
Converting Characters	19-11
Changing the EOL Sequence	19-15
Parity Generation and Checking	19-16
Determining the Outcome of ASSIGN Statements	19-18

20. Transfers and Buffered I/O

The Purpose of Transfers	20-1
Overview of Buffers and Transfers	20-2
Inbound and Outbound Transfers	20-2
Supported Transfer Sources and Destinations	20-3
Interfaces	20-3
File Types	20-4
Pipes (BASIC/UX only)	20-4
Transfer Examples	20-5
A Closer Look at Buffers	20-6
Types of Buffers	20-6
Creating Named Buffers	20-6

Assigning I/O Path Names to Named Buffers	20-7
Assigning I/O Path Names to Unnamed Buffers	20-7
Buffer-Type Registers	20-8
Buffer Size Register	20-8
Buffer Life Time	20-8
Buffer Pointers	20-9
Accessing Named Buffers via Variable Names	20-11
A Closer Look at Transfers	20-12
Transfer Methods for BASIC/WS	20-13
Transfer Methods for BASIC/DOS	20-13
Transfer Methods for BASIC/UX	20-13
OUTPUT and ENTER and Buffers	20-13
Transfer Formatting	20-14
Transfer Termination	20-14
Visually Determining Transfer Status	20-14
Choosing Transfer Parameters	20-14
Continuing Transfers Indefinitely	20-15
Waiting for a Transfer to End (Non-Overlapped Transfers)	20-15
Continuous Non-Overlapped Transfers	20-16
Transferring a Specified Number of Bytes	20-16
Delimiter Characters	20-16
Using the END Indication with Transfers	20-16
Transferring Records	20-17
Multiple Termination Conditions	20-17
TRANSFER Records and Termination	20-18
Transfer Event-Initiated Branching	20-19
Terminating a Transfer	20-20
More Transfer Examples	20-22
Special Considerations	20-26
Transfer with Care	20-26
Statements Which Affect Concurrency	20-27
Error Reporting	20-29
Suspended Transfers	20-30
Transfer Performance	20-30
Sector Size	20-30
Internal Disk Drives of Models 226 and 236 Computers	20-31
Overlapped Transfers and Disk Drives	20-31
Transfer Methods and Rates	20-34

Available Methods	20-34
DMA Mode	20-34
INT Mode	20-35
Burst Interrupt Mode	20-35
Approximate Transfer Rates for Devices	20-35
Using TRANSFER in BASIC/UX	20-36
Locking an Interface During a TRANSFER	20-36
Using the Burst I/O Mode During a TRANSFER	20-37
Transfer Methods for BASIC/UX	20-37
Timing Issues	20-38
TRANSFER Restrictions	20-39
All TRANSFERS Use Only One BUFFER	20-39
Devices and Files Not Supported for TRANSFER	20-39
Number of Simultaneous TRANSFERS	20-40
Interactions with Other Keywords	20-40
GPIO	20-40
Serial	20-40
Datacomm	20-41
HP-IB	20-41
Premature Termination	20-42
Changing Buffer Attributes	20-42
Buffer Status and Control Registers	20-43
21. Interface Overview	
The HP-IB Interface	21-1
Initial Installation	21-2
Communicating with Devices	21-2
HP-IB Device Selectors	21-2
Moving Data Through the HP-IB	21-3
General Structure of the HP-IB	21-3
Examples of Bus Sequences	21-5
Addressing Multiple Listeners	21-6
Secondary Addressing	21-7
General Bus Management	21-7
Remote Control of Devices	21-8
Locking Out Local Control	21-9
Enabling Local Control	21-9
Triggering HP-IB Devices	21-10

Clearing HP-IB Devices	21-10
Aborting Bus Activity	21-11
HP-IB Service Requests	21-11
Setting Up and Enabling SRQ Interrupts	21-12
Servicing SRQ Interrupts	21-12
Polling HP-IB Devices	21-12
Configuring Parallel Poll Responses	21-13
Conducting a Parallel Poll	21-13
Disabling Parallel Poll Responses	21-13
Conducting a Serial Poll	21-14
The Computer As a Non-Active Controller	21-14
Determining Controller Status and Address	21-14
Status Register 3: Controller Status and Address	21-15
Changing the Controller's Address	21-15
Passing Control	21-16
Interrupts While Non-Active Controller	21-16
Status Register 5: Interrupt Enable Mask	21-18
Requesting Service	21-21
Responding to Parallel Polls	21-21
Responding to Serial Polls	21-22
HP-IB Control Lines	21-23
Handshake Lines	21-24
The Attention Line (ATN)	21-24
The Interface Clear Line (IFC)	21-25
The Remote Enable Line (REN)	21-25
The End or Identify Line (EOI)	21-25
The Service Request Line (SRQ)	21-25
References	21-26
The RS-232 Serial Interface	21-26
Asynchronous Data Communication	21-27
Character Format	21-27
Parity	21-28
Error Detection	21-29
Data Transfers Between Computer and Peripheral	21-30
Overview of Serial Interface Programming	21-30
Initializing the Interconnection	21-31
Determining Operating Parameters	21-31
Hardware Parameters	21-31

Character Format Parameters	21-31
Using Interface Defaults to Simplify Programming	21-32
Using Program Control to Override Defaults	21-32
Interface Reset	21-32
Selecting the Baud Rate	21-33
Setting Character Format and Parity	21-33
Data Transfers	21-34
Program Flow	21-34
Data Output	21-34
Data Entry	21-34
Modem Line Handshaking	21-35
Incoming Data Error Detection and Handling	21-35
Trapping Serial Interface Errors	21-35
The GPIO Interface	21-36
Interface Description	21-36
Interface Configuration	21-37
Interface Select Code	21-37
Hardware Interrupt Priority	21-37
Data Logic Sense	21-37
Data Handshake Methods	21-38
The GPIO handshakes data with three signal lines	21-38
Handshake Logic Sense	21-38
Handshake Modes	21-39
Data-In Clock Source	21-39
Optional Peripheral Status Check	21-39
Interface Reset	21-40
Using OUTPUT and ENTER Through the GPIO	21-40
ASCII and Internal Representations	21-41
Example Statements Using OUTPUT	21-41
Example Statements Using ENTER	21-41
Example Statements that Output Data Words	21-42
Example Statements that Enter Data Words	21-42
GPIO Timeouts	21-43
Timeout Time Parameter	21-43
Timeout Service Routines	21-43
GPIO Interrupts	21-44
Types of Interrupt Events	21-44
Setting Up and Enabling Events	21-44

Interrupt Enable Register: (ENABLE INTR)	21-45
Interface Ready	21-45
External Interrupt Request	21-45
Interrupt Service Routines	21-46
Status Register 4: Interface Ready	21-46
Status Register 5: Peripheral Status	21-46

Index

Introduction

This manual is intended to introduce you to the HP Series 200/300 BASIC programming language, commonly known as “HP BASIC,” and to provide some helpful hints on getting the most utility from it. Although this manual assumes that you have had some previous programming experience, you need not have a high skill level, nor does your previous experience need to be in BASIC. If you have never programmed a computer before, it will probably be more comfortable for you to start with one of the many beginner’s text books available from various publishing companies. However, some beginners may find that they are able to start in this manual by concentrating on the fundamentals presented in the first few chapters. If you are a programming expert or are already familiar with the BASIC language of other HP computers, you may start faster by going directly to the *HP BASIC Language Reference* and checking the keywords you normally use. You can always come back to this manual when you have extra time to explore the computer’s capabilities, or if you bump into an unfamiliar concept.

After reading each section and trying the examples shown, try your own examples. Experiment. You cannot damage the computer by pressing the wrong keys. The worst thing that can happen is that an error message will appear. All errors are listed in the “Error Messages” section in volume 2 of the *HP BASIC Language Reference*.

Some Important Notations

There are three implementations of HP BASIC, which are described below. The notations BASIC/WS, BASIC/UX, and BASIC/DOS are used throughout this manual to identify these implementations.

- BASIC/WS** The Workstation implementation, which is a combined language and operating system that runs on HP 9000 Series 200/300 computers. (This is probably the most familiar implementation of HP Series 200/300 BASIC language.)
- BASIC/UX** The HP-UX implementation, which is essentially the BASIC interpreter and part of the BASIC Workstation operating system that runs as a set of processes “on top of” the HP-UX operating system.
- BASIC/DOS** The DOS implementation, which is essentially Workstation BASIC modified slightly to run on the HP Measurement Coprocessor. BASIC/DOS supports both the HP 82300 Measurement Coprocessor and the HP 82324 High-Performance Measurement Coprocessor. These coprocessors provide HP Series 200/300 computer architecture on a PC plug-in card.

Most of the programming techniques described in this manual are applicable to all three implementations of HP BASIC. However, where there are specific differences, they will be identified.

What's In This Manual?

No matter what your skill level, it is helpful to understand the contents and organization of this manual. First of all, there are some things that it is *not*. Because it is organized by topics and concepts, it is not a good place to find an individual keyword in a hurry. Keywords can be found using the index, but even so, they are often embedded in discussions, contained in more than one place, or only partially explained. Also, this is not a good place to find complete syntactical details. Program statements are often presented only in the form that applies to the specific concept being discussed, even though there

1-2 Introduction

may be other forms of the statement that accomplish different purposes. If you want to quickly find the complete formal syntax of a keyword, use the *HP BASIC Language Reference*. It is specifically intended for that purpose.

This manual contains explanations and programming hints organized topically. A program performs various sub-tasks as it completes its overall job. Many of these tasks should be viewed separately to be understood more easily and used more effectively. For example, perhaps you have experience in another programming language. You know exactly what a “loop” does, but you didn’t find the statement you were looking for in the *HP BASIC Language Reference*. In the chapter on “Program Structure and Flow,” there is a section called “Repetition” which explains the kinds of loops available and all the statements needed to create them. The following is an overview of the chapters in this manual. (Note that the *HP BASIC 6.2 Advanced Programming Techniques* manual provides additional information about selected programming topics. References to this manual are given where appropriate.)

Chapter 1: Introduction

Chapter 2: Program Structure and Flow

This chapter tells how the computer finds its way around your program and offers ideas on getting it to follow the proper path efficiently.

Chapter 3: Numeric Computation

This chapter covers mathematical operations and the use of numeric variables. It includes discussions on types of variables, expression evaluation, arrays, and methods of managing data memory.

Chapter 4: Numeric Arrays

This chapter covers numeric array operations. (The *HP BASIC 6.2 Advanced Programming Techniques* manual provides additional information about this topic.)

Chapter 5: String Manipulation

Although string data can be used for any purpose the programmer desires, it is most often used for the processing of characters, words, and text. This chapter explains the programming tools available for processing string data. (The *HP BASIC 6.2 Advanced Programming Techniques* manual provides additional information about this topic.)

Chapter 6: Subprograms and User-Defined Functions

An outstanding feature of this language is its ability to change program contexts and the speed with which it can do so. Alternate contexts (or environments) are available as user-defined functions or subprograms. These are discussed in this chapter.

Chapter 7: Data Storage and Retrieval

This chapter shows many of the alternatives available for storing the data that is intended as program input or created as program output. (The *HP BASIC 6.2 Advanced Programming Techniques* manual provides additional information about file I/O.)

Chapter 8: Graphics Techniques

This chapter introduces you to the powerful set of graphics statements in HP BASIC and teaches you how to use them to produce pleasing output. (The *HP BASIC 6.2 Advanced Programming Techniques* manual provides additional information about various graphics topics.)

Chapter 9: Graphics on Printers and Plotters

In contrast to Chapter 8 which describes how to generate graphic images on your CRT display, this chapter explains how to select external printing and plotting devices you can use to generate graphics on paper.

Chapter 10: Using a Printer

This chapter tells how to use an external printer. Also covered are the formatting techniques (useful on both printer and CRT) to create organized, highly-readable printouts.

Chapter 11: Using the Clock and Timers

An accurate real-time clock is available with timing resolution to the hundredth of a second and a range of years. Its capabilities are covered in this chapter.

Chapter 12: Handling Errors

This chapter discusses techniques for intercepting (or trapping) errors that might occur while a program is running. Many errors can be dealt with easily by a programmer. Error trapping keeps the program running and provides valuable assistance to the computer operator.

Chapter 13: Debugging Programs

We all wish that every program would run perfectly the first time and every time. Unfortunately, there is little evidence in real life to support that fantasy. The next best thing is to have debugging tools. This chapter explains the powerful and convenient debugging features available on the computer.

Chapter 14: Introduction to I/O

This chapter introduces the concepts of computer interfaces and Input/Output (I/O) operations. Chapters 15 through 21 cover specific I/O programming techniques. (The *HP BASIC 6.2 Advanced Programming Techniques* manual provides additional information about I/O programming techniques.)

Chapter 15: Outputting Data

This chapter describes free-form and formatted output operations using the OUTPUT statement.

Chapter 16: Entering Data

This chapter describes free-form and formatted input operations using the ENTER statement.

Chapter 17: Registers

This chapter tells how to address interface registers and I/O path registers using the STATUS and CONTROL statements.

Chapter 18: Interrupts and Timeouts

This chapter tells how to handle interface interrupts and timeouts in programs.

Chapter 19: I/O Path Attributes

This chapter describes programming techniques involving the FORMAT attribute and other I/O path attributes.

Chapter 20: Transfers and Buffered I/O

This chapter describes I/O buffers and the use of the TRANSFER statement.



Chapter 21: Interface Overview

This chapter gives an overview of the HP-IB, RS-232 Serial, and GPIO interfaces, and describes specific programming techniques for each. For additional information about these interfaces and other available interfaces for HP 9000 Series 200/300 computers, refer to the *HP BASIC 6.2 Interface Reference* manual.

Program Structure and Flow

Two of the most significant characteristics of a computer are its ability to perform computations and its ability to make decisions. If the execution sequence could never be changed within a program, the computer could do little more than plug numbers into a formula. Computers have powerful computational features, but the heart of a computer's intelligence is its ability to make decisions. This chapter discusses the ways that decisions are used in controlling the flow of program execution.

2 The Program Counter

The **program counter** is the part of the computer's internal system that tells it which line to execute. Unless otherwise specified, the program counter automatically updates at the end of each line so that it points to the next program line.

This fundamental type of program flow is called "linear flow" and is illustrated in the following example. With this example, visualize the flow of statement execution as being a straight line through the program listing.

Program Lines	Value in Program Counter at End of Line
120 R=R+2	130
130 Area=PI*R^2	131
131 PRINT R	140
140 PRINT "Area =";Area	150
150 STOP	don't care

Although linear flow seems very elementary, always remember that this is the computer's normal mode of operation.

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

- Sequence
- Selection (conditional execution)
- Repetition

In addition to capabilities in all three of these categories, your computer also has a powerful special case of selection, called **event-initiated branching**. The rest of this chapter shows how to use all of these types of program flow and gives suggestions for choosing the type of flow that is best for your application.

Sequence

This section describes the types of sequences of program execution:

- Linear flow—the BASIC system executes lines in sequential fashion.
- Halting program execution—stopping the flow of a program.
- Branching—the BASIC program redirects the normally sequential flow.

Linear Flow

The simplest form of sequence is linear flow. The preceding section showed an example of this type of flow. Although linear flow is not at all glamorous, it has a very important purpose. Most operations required of the computer are too complex to perform using one line of BASIC. Linear flow allows many program lines to be grouped together to perform a specific task in a predictable manner. Although this form of flow requires little explanation, keep these characteristics in mind:

- Linear flow involves *no* decision making. Unless there is an error condition, the program lines involved in this type of flow will always be executed in exactly the same order, regardless of the results of, or arguments to, any expression.
- Linear flow is the default mode of program 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

One of the obvious alternatives to executing the next line in sequence is not to execute anything. There are three statements that can be used to block the execution of the next line and halt program flow. Each of these statements has a specific purpose, as explained in the following paragraphs.

A main program is a list of program lines with an END statement on the last line. Marking the end of the main program is the primary purpose of the END statement. Therefore, a program can contain only one END statement. The secondary purpose of the END statement is stopping program execution. When

an END statement is executed, program flow stops and the program moves into the stopped (non-continuable) state.

It is often necessary to stop the program flow at some point other than the end of the main program. This is the purpose of the STOP statement. A program can contain any number of STOP statements in any program context. When a STOP statement is executed, program flow stops and the program moves into the stopped (non-continuable) state. Also, if the STOP statement is executed in a subprogram context, the main program context is restored. (Subprograms and context switching are explained in the “User-Defined Functions and Subprograms” chapter.)

As an example of the use of STOP and END, consider the following program.

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

When the **RUN** key (**f3** in the System menu of ITF keyboards) is pressed, the computer prints 31 on the CRT and the Run Indicator (lower right corner of CRT) goes off. This first press of the **RUN** key caused linear execution of lines 100 thru 130, with line 130 stopping that execution. If the **RUN** key is pressed again, the same thing will happen; the program does *not* resume execution from its stopping point in response to a RUN command. However, RUN can specify a starting point. So, execute RUN 140. The computer prints 0 and stops. This command caused linear execution of lines 140 thru 160, with line 160 stopping that execution. However, a RUN command also causes a prerun initialization which zeroed the value of the variable Radius.

You could try pressing **CONTINUE** or **CONT** (**f2** in the System menu of ITF keyboards) in the preceding example, but you will get an error. A stopped program is not continuable. This leads up to the third statement for halting program flow. Replace the STOP statement on line 130 with a PAUSE statement, yielding the following program.

2-4 Program Structure and Flow

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

Now when the program is run, and the computer prints 31 on the CRT. Then when **CONTINUE** is pressed, the computer prints 78 on the CRT. The purpose of the PAUSE statement is to *temporarily* halt program execution, leaving the program counter intact and the program in a continuable state. One common use for the PAUSE statement is in program troubleshooting and debugging. This is covered in the “Program Debugging” chapter. Another use for PAUSE is to allow time for the computer user to read messages or follow instructions. Interfacing with a human is covered in greater depth in the “Communicating with the Operator” chapter, but here is one example of using the PAUSE statement in this way.

```
100 PRINT "This program generates a cross-reference"
110 PRINT "printout. The file to be cross-referenced"
120 PRINT "must be an ASCII file containing a BASIC"
130 PRINT "program."
140 PRINT
150 PRINT "Insert the disk with your files on it and"
160 PRINT "press CONTINUE."
170 PAUSE
180 !   Program execution resumes here after CONTINUE
```

Lines 100 thru 160 are instructions to the program user. Since a user will often just load a program and run it, the programmer cannot assume that the user's disk is in place at the start of the program. The instructions on the CRT remind the user of the program's purpose and review the initial actions needed. The PAUSE statement on line 170 gives the user all the time he needs to read the instructions, remove the program disk, and insert the “data disk.” It would be ridiculous to use a WAIT statement to try to anticipate the number of seconds required for these actions. The PAUSE statement gives freedom to the user to take as little or as much time as necessary.

When **CONTINUE** (**F2**) is pressed, the program resumes with any necessary input of file names and assignments. Questions such as “Have you inserted the

proper disk?” are unnecessary now. The user has already indicated compliance with the instructions by pressing `CONTINUE`.

Simple Branching

An alternative to linear flow is branching. Although conditional branching is one of the building blocks for selection structures, the unconditional branch is simply a redirection of sequential flow. The keywords which provide unconditional branching are `GOTO`, `GOSUB`, `CALL`, and `FN`. The `CALL` and `FN` keywords invoke new contexts, in addition to their branching action. This is a complex action that is the topic of the “Subprograms and User-Defined Functions” chapter. This section discusses the use of `GOSUB` and `GOTO`.

Using `GOTO`

First, you should be aware that the structuring capabilities available in BASIC make it possible to avoid the use of the unconditional `GOTO` in most applications. You should also be aware that this is a highly-desirable goal. (See the section on “Top-Down Design” in the “User-Defined Functions and Subprograms” chapter.)

The only difference between linear flow and a `GOTO` is that the `GOTO` loads the program counter with a value that is (usually) different from the next-higher line number. The `GOTO` statement can specify either the line number or the line label of the destination. The following example shows the program flow and contents of the program counter in a program segment containing a `GOTO`.

Program Lines	Value in Program Counter at End of Line
180 R=R+2	190
190 Area=PI*R^2	200
200 GOTO 240	240
210 Width=Width+1	220
220 Length=Length+1	230
230 Area=Width*Length	240
240 PRINT "Area =";Area	250
250 GOTO 210	210

As you can see, the execution is still sequential and no decision-making is involved. The first GOTO (line 200) produces a forward jump, and the second GOTO (line 250) produces a backward jump. A forward jump is used to skip over a section of the program. An unconditional backward jump can produce an **infinite loop**. This is the endless repetition of a section of the program. In this example, the infinite loop is line 210 thru 250.

An infinite loop by itself is not usually a desirable program structure. However, it does have its place when mixed with conditional branching or event-initiated branching. Examples of these structures are given later in this chapter.

Using GOSUB

The GOSUB statement is used to transfer program execution to a subroutine. Note that a subroutine and a subprogram are very different in HP BASIC. Calling a **subprogram** invokes a new context. Subprograms can declare formal parameters and local variables. A **subroutine** is simply a segment of a program that is entered with a GOSUB and exited with a RETURN. Subroutines are always in the same context as the program line that invokes them. There are no parameters passed and no local variables. If you are a newcomer to HP's BASIC, be careful to distinguish between these two terms. They have been used differently in some other programming languages.

The GOSUB is very useful in structuring and controlling programs. The similarity it has to a procedure call is that program flow can automatically return to the proper line when the subroutine is finished. The GOSUB statement can specify either the line label or the line number of the desired subroutine entry point. The following example shows the program flow and contents of the program counter in a program segment containing a GOSUB.

Program Lines	Value in Program Counter at End of Line
300 R=R+2	310
310 Area=PI*R^2	320
320 GOSUB 1000	1000 (see below)
330 Width=Width+1	340
340 Length=Length+1	350
350 ! Program continues	
.	
.	
.	
1000 PRINT Area;"square in."	1010
1010 Cent=Area*6.4516	1020
1020 PRINT Cent;"square cm"	1030
1030 PRINT	1040
1040 RETURN	330

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 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.

2-8 Program Structure and Flow

```

10 Start:      ! Program begins here.
20             PRINT "Hello"
30             GOTO Finish
40             DISP "Stop"           ! This line is never executed.
50 Finish:     PRINT "Goodbye"      ! PRINT is the 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 statement PRINT, and ends with a comment. This program simply prints “Hello” and “Goodbye” repeatedly until the program is paused. 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**. As the name implies, a certain segment of the program either is or is not executed according to the results of a test or condition. This is the basic action which gives the computer an appearance of possessing intelligence. Actually, it is the intelligence of the programmer which is remembered by the program and reflected in the pattern of conditional execution.

This section presents the conditional-execution statements according to various applications.

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 a numeric expression that is evaluated as being either true or false. If true (non-zero), the conditional segment is executed. If false (zero), the conditional segment is bypassed. Although the expression contained in an IF ... THEN is treated as a Boolean expression, note that there is no "BOOLEAN" data type. Any valid numeric expression is allowed.

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

```
100 IF Ph>7.7 THEN OUTPUT Valve USING "#,B";0
```

Notice the test (Ph>7.7) and the conditional statement (OUTPUT Valve..) which appear on either side of the keyword THEN. When the computer executes this program line, it evaluates the expression Ph>7.7. If the value contained in the variable Ph is 7.7 or less, the expression evaluates to 0 (false), and the line is exited. If the value contained in the variable Ph is greater than 7.7, the expression evaluates as 1 (true), and the OUTPUT statement is executed. If you don't already understand logical and relational operators, refer to the "Numeric Computation" and "String Manipulation" chapters. (Although "pH" is the correct chemical expression, it is not valid as a BASIC variable name and thus, "Ph" has been substituted. For further information about naming variables, refer to "Naming Variables" in chapter 3.)

By the way, the image specifier #,B causes the output of a single byte. In the example, the value for that byte is specified as zero (all bits cleared). Presumably, this turns off all devices connected to a GPIO (digital) interface.

Note that the same variable is allowed on both sides of an IF ... THEN statement. For example, the following statement could be used to keep a user-supplied value within bounds.

2-10 Program Structure and Flow

```
IF Number>9 THEN Number=9
```

When the computer executes this statement, it checks the initial value of **Number**. If the variable contains a value less than or equal to nine, that value is left unchanged, and the statement is exited. If the value of **Number** is greater than nine, the conditional assignment is performed, replacing the original value in **Number** with the value nine.

Prohibited Statements

Certain statements are not allowed as the conditional statement in a single-line IF ... THEN. The disallowed statements are used for various purposes, but the "common denominator" is that the computer needs to find them during prerun as the first keyword on a line.

Multiple-Line Conditional Segments

If the conditional program segment requires more than one statement, a slightly different structure is used. Let's expand the valve-control example.

```
100 IF Ph>7.7 THEN
110   OUTPUT Valve USING "#,B";0
120   PRINT "Final Ph =";Ph
130   GOSUB Next_tube
140 END IF
150 ! Program continues here
```

Any number of program lines can be placed between a THEN and an END IF statement. In executing this example, the computer evaluates the expression **Ph>7.7** in the IF ... THEN statement. If the result is false, the program counter is set to 150, and execution resumes with the line following the END IF statement. If the condition is true, the program counter is set to 110, and the three conditional statements (lines 110, 120, 130) are executed. Program flow then picks up at line 150, because the END IF is only used during prerun.

When using multiple-line IF ... THEN structures, remember to mark the end of the structure with an END IF statement and don't put any of the statements on the same line as the IF ... THEN. If the beginning and end of the structure are not properly marked, the computer reports error 347 during prerun.

The conditional segment can contain any statement except one that is used to set context boundaries (such as END or DEF FN). In the previous example,

the `GOSUB Next_tube` could have been a `GOTO Next_tube`. In that case, program execution does not pass through line 150 when the condition is true. A false condition would cause a branch to line 150, while a true condition would send execution from line 100, to 110, to 120, to 130, and then to the line labeled "Next_tube."

If structuring statements are used within a multiple-line `IF ... THEN`, the entire structure must be contained in one conditional segment. These are **nested** constructs. The following example shows some properly nested constructs. Notice that the use of indenting improves the readability of the code.

```

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

```

Choosing One of Two Segments

Often you want a program flow that passes through only one of two paths depending upon a condition. For example, in the following example program, if `Flag = 1`, then the program flow at the end of line 420 would pass directly to line 480. However, if `Flag = 0`, then the program flow at line 400 would pass directly to line 430. If you have ever been forced to program this type of structure using only the conditional `GOTO`, you know that the result is much more confusing than it needs to be.

```

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 language has an `IF ... THEN ... ELSE` structure which makes the one-of-two choice easy and readable. The following example looks at a device

2-12 Program Structure and Flow

selector which may or may not contain a primary address. The variable `Isc` is needed later in the program and must be only an interface select code. If the operator-supplied device selector is greater than 31, the interface select code is extracted from it. If it is equal to or less than 31, it already is an interface select code. (This example assumes that no secondary addressing is used.)

```

500 IF Select>31 THEN
510   Isc=Select DIV 100
520 ELSE
530   Isc=Select
540 END IF

```

Notice that this structure is similar to the multiple-line `IF ... THEN` shown previously. The only difference is the addition of the keyword `ELSE`. Like the previous example, the structure is terminated by `END IF`, and the proper nesting of other structures is allowed.

Choosing One of Many Segments

Using SELECT

Consider as an example the processing of readings from a voltmeter. In this example, we assume that the reading has already been entered, and it contained a function code. These hypothetical function codes identify the type of reading and are shown in the following table.

Function Codes

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

The first example shows the use of the `SELECT` construct. The function code is contained in the variable `Funct$`. For the sake of simplicity, the example does not show any actual processing. Comments are used to identify the location of the processing segments. The rules about illegal statements and proper nesting are the same as those discussed previously in the `IF ... THEN` section.

```

2000 SELECT Funct$
2010 CASE "DV"
2020     !
2030     ! Processing for DC Volts
2040     !
2050 CASE "DI"
2060     !
2070     ! Processing for DC Amps
2080     !
2090 CASE "OM"
2100     !
2110     ! Processing for Ohms
2120     !
2130 CASE ELSE
2140     BEEP
2150     PRINT "INVALID READING"
2160 END SELECT
2170 ! Program execution continues here

```

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

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

```

CASE 1 TO 31    Processing for interface select code

CASE -1,1,3 TO 7,>15  Specifying multiple matches

CASE CHR$(27)&"@""&Eol$    Using a string expression

```

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 a SELECT statement actually means that there was an error in that line *or* in one of the CASE statements.

2-14 Program Structure and Flow

Repetition

Humans usually prefer tasks with variety that avoid tedious repetition. A computer does not have this shortcoming. You have 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 ... END 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. With this structure, a program segment is executed a predetermined number of times. The FOR statement marks the beginning of the repeated segment and establishes the number of repetitions. The NEXT statement marks the end of the repeated segment. This structure uses a numeric variable as a **loop counter**. This variable is available for use within the loop, if desired.

The number of loop iterations is determined by the FOR statement. This statement identifies the loop counter, assigns a starting value to it, specifies the desired final value, and determines the step size that will be used to take the loop counter from the starting value to the final value. For example, in the following FOR ... NEXT loop, the starting value is 10, the final value is 0, and the step size is -1. The repeated segment is lines 210 through 240.

```
200 FOR Count=10 TO 0 STEP -1
210     BEEP
220     PRINT Count
230     WAIT 1
240 NEXT Count
```

When the loop counter is an INTEGER, the number of iterations can be predicted using the following formula:

$$INT\left(\frac{StepSize + FinalValue - StartingValue}{StepSize}\right)$$

Note that the formula applies to the values in the variables, not necessarily the numbers in the program source. For example, if you use an INTEGER loop counter and specify a step size of 0.7, the value will be rounded to one. Therefore, 1 should be used in the formula, not 0.7.

The loop counter can be a REAL number, with REAL quantities for the step size, starting, or final values. In some cases, using REAL numbers will cause the number of iterations to be off by one from the preceding formula. This is because the NEXT statement performs an “increment and compare,” and there is a slight inaccuracy in the comparison of REAL numbers. If you are interested, this is discussed in the next chapter. However, there is no clean way around it with FOR ... NEXT loops. Here is an example:

```
200 Count=0
210 FOR X=10 TO 20
220   Count=Count+1
230   PRINT Count
240 NEXT X
```

According to the formula, this loop should execute 11 times: $\text{INT}((1+20-10)/1=11)$. The result on the CRT confirms this when the loop is executed. If line 210 is changed to:

```
210 FOR X=1 TO 2 STEP .1
```

the formula still yields 11 as the number of iterations. However, executing the loop produces only 10 repetitions. This is because of a very, very small accumulated error that results from the successive addition of one-tenth. The error is less significant than the 15th digit, but discernable to the computer. In this case, rounding cannot be performed at a time that would help. When you find yourself in this situation, one way out is to add a slight adjustment factor to the final value. The following line does give the 11 iterations predicted by the formula.

```
210 FOR X=1 TO 2.05 STEP .1
```

Remembering the “increment and compare” operation at the bottom of the loop is helpful. After the loop counter is updated, it is compared to the final value established by the FOR statement. If the loop counter has *passed* the specified final value, the loop is exited. If it has *not passed* the specified final value, the loop is repeated. The loop counter retains its exit value after the loop is finished. This is not necessarily one full step past the final value. For example: FOR I=1 TO 9.9.

2-16 Program Structure and Flow

Conditional Number of Iterations

The FOR ... NEXT loop produces a fixed number of iterations, established by the FOR statement before the loop is executed. Some applications need a loop that is executed until a certain condition is true, without specifically stating the number of iterations involved. Consider a very simple example. The following segment asks the operator to input a positive number. Presumably, negative numbers are not acceptable. A looping structure is used to repeat the entry operation if an improper value is given. Notice that it is not important *how many times* the loop is executed. If it only takes once, that is just fine. If the poor operator takes ten tries before he realizes what the computer is asking for, so be it. What is important is that a *specific condition* is met. In this example, the condition is that a value be non-negative. As soon as that condition has been satisfied, the loop is exited.

```

800 REPEAT
810   INPUT "Enter a positive number",Number
820 UNTIL Number>=0

```

A typical use of this is an iterative problem involving non-linear increments. One example is musical notes. Performing the same operation on all the notes in a 3-octave band is a repetitive process, but not a linear one. Musical notes are related geometrically by the 12th root of two. The following example simply prints the frequencies involved, but your application could involve any number of operations.

```

1200 Note=110      ! Start at low A
1210 REPEAT
1220   PRINT Note;
1230   Note=Note*2^(1/12)
1240 UNTIL Note>880 ! End at high A

```

For this example, a FOR ... NEXT loop might have been used, with the loop counter appearing in an exponent. That would work because it is relatively easy to know how many notes there are in three octaves of the musical scale. However, the REPEAT ... UNTIL structure is more flexible than FOR ... NEXT when working with exponential data in general. Examples often occur in the area of graphics. The following segment could be used to plot audio frequency data, where the x-axis is logarithmic.

```

1500 Freq=20
1510 MOVE LOG(Freq),FNFunction(Freq)
1520 REPEAT
1530     DRAW LOG(Freq),FNFunction(Freq)
1540     Freq=Freq*1.2
1550 UNTIL Freq>20000

```

The WHILE loop is used for the same purpose as the REPEAT loop. The only difference between the two is the location of the test for exiting the loop. The REPEAT loop has its test at the bottom. This means that the loop is always executed at least once, regardless of the value of the condition. The WHILE loop has its test at the top. Therefore, it is possible for the loop to be skipped entirely (if the conditions so dictate). The following segment shows the same plotting example using a WHILE loop.

```

1500 Freq=20
1510 MOVE LOG(Freq),FNFunction(Freq)
1520 WHILE Freq<=20000
1530     DRAW LOG(Freq),FNFunction(Freq)
1540     Freq=Freq*1.2
1550 END WHILE

```

The REPEAT ... UNTIL and WHILE structures are especially useful for tasks that are impossible with a FOR ... NEXT loop. One such situation is a loop where both the loop counter and the final value are changing. Consider the example of stripping all control characters from a string. This can't be done in a loop that starts FOR I=1 TO LEN(A\$), because the length of A\$ changes each time a character is deleted. Therefore, the loop counter used as a subscript will eventually exceed the length of the string by more than one, generating an error. The WHILE loop does not have this problem. Note that the test at the top of the loop prevents the subscripting from being attempted on a null string. This is necessary to avoid an error.

```

600 I=1
610 WHILE I<=LEN(A$)
620     IF A$[I;1]<CHR$(32) THEN
630         A$[I]=A$[I+1]
640     ELSE
650         I=I+1
660     END IF
670 END WHILE

```

2-18 Program Structure and Flow

Arbitrary Exit Points

A pass through any of the loop structures discussed so far included the entire program segment between the top and the bottom of the loop. There are times when this is not the desired program flow. The LOOP structure defines the repeated program segment and allows any number of conditional exits points in that segment.

For the first example, consider a search-and-replace operation on string data. In this example, the “shift out” control character is being used to initiate underlining on a printer that understands standard escape sequences. The “shift in” control character is used to turn off the underline mode. (There is nothing significant about this choice of characters. any combination of characters could serve the same purpose.)

One approach is to use a loop to search every character in every string to see if it is one of the special characters. There are two problems with this method. First, it is a little cumbersome when the replacement string is a different length than the target string. Second, it is slow. Admittedly, speed is not a significant consideration when driving common mechanical printers. But the destination might eventually be a laser printer or mass storage file, making the program’s speed more visible.

A better approach is to use the POS function to locate the target string. Since this function locates only the first occurrence of a pattern, it must be placed in a loop to insure that multiple occurrences will be found. The LOOP structure is well suited to this task, as shown in the following example.

```

2000 LOOP
2010   Position=POS(A$,CHR$(14))
2020 EXIT IF NOT Position
2030   A$[Position]=CHR$(27)&"&dD"&A$[Position+1]
2040 END LOOP
2050 !
2060 LOOP
2070   Position=POS(A$,CHR$(15))
2080 EXIT IF NOT Position
2090   A$[Position]=CHR$(27)&"&d@"&A$[Position+1]
2100 END LOOP
2110 ! Last EXIT goes to here

```

In this segment, all occurrences of “shift out” are replaced by “escape &dD” to enable underline mode. All occurrences of “shift in” are replaced by “escape

&d@" to disable underlining. Notice that there is no problem replacing one character with four (assuming that A\$ is large enough). Lines containing no special characters are processed by only two POS functions, which is much faster and cleaner than performing two comparisons for every character in every line.

Another common use for this structure is the processing of operator input. Recall the REPEAT ... UNTIL example that tested for the input of a positive number. Although this structure kept the computer happy, it left the operator in the dark. The LOOP structure provides for the additional processing needed, as shown in the following example.

```
200 LOOP
210   INPUT "Enter a positive number.",Number
220   EXIT IF Number>=0
230   BEEP
240   PRINT
250   PRINT "Negative numbers are not allowed."
260   PRINT "Repeat entry with a positive number."
270 END LOOP
```

Another point to remember is that the LOOP structure permits more than one exit point. This allows loops that are exited on a "whichever comes first" basis. Also, the EXIT IF statement can be at the top or bottom of the loop. This means that the LOOP structure can serve the same purposes as REPEAT ... UNTIL and WHILE ... END WHILE, if that suits your programming style.

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

WRONG:

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

RIGHT:

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

Event-Initiated Branching

Your computer has a special kind of program flow that provides some very powerful tools. This tool, called **event-initiated branching**, uses interrupts to redirect program flow. The process can be visualized as a special case of selection. Every time program flow leaves a line, the computer executes an “event-checking” routine. This is set of machine-language “if ... then” statements concerning interrupts. If an event is:

- Enabled to initiate a branch (with an ON-event statement)
- The event occurs

Then this “event-checking” routine causes the program to branch (as specified in the ON-event statement).

The process of “event checking” is represented in the following line. Notice that it is possible for event-initiated branching to occur at the end of any

program line, which includes the lines of a subprogram. This can give the appearance of “middle-of-line” branching when it occurs during a user-defined function.

```
100 X=Radius*FNMy_function/COS(Angle)^Exponent
```

In the above example, the branch may occur while FNMy_function is being executed.

In the following example, these potential branching points are marked by the words `gosub event_check`. This does not refer to a BASIC subroutine, but is just a symbolic reminder of where event-initiated branching can occur. If the operating system finds an event has occurred (and the corresponding branch is currently enabled), then a branch is initiated. If not, program execution resumes with the “normal” program flow.

```
10 PRINT X ! gosub event_check
20 X=X+1 ! gosub event_check
30 GOTO 10 ! gosub event_check
```

Types of Events

Event-initiated branching is established by the `ON..event` statements. Here is a list of the statements that fall in this category, along with the corresponding event that causes a branch:

ON CDIAL	an interrupt generated by turning a knob—a rotary pulse generator—on a Control Dial box (see the “Communicating with the Operator” chapter of the <i>HP BASIC 6.2 Advanced Programming Techniques</i> manual).
ON CYCLE	cyclical (periodic, repetitive) interrupts from the clock (see the “Clock and Timers” chapter of this manual)
ON DELAY	a one-time interrupt from the clock (see the “Clock and Timers” chapter of this manual)
ON END	an interrupt upon reaching an end-of-file (EOF) condition (see the “Data Storage and Retrieval” chapter of this manual)
ON ERROR	an interrupt when a run-time error is encountered (see the “Handling Errors” chapter of this manual)

2-22 Program Structure and Flow

ON EOR	an interrupt when an end-of-record is encountered during a TRANSFER statement (see the “Data Storage and Retrieval” chapter of this manual)
ON EOT	an interrupt when an end-of-TRANSFER condition is encountered (see chapter 20, “Transfers and Buffered I/O.”)
ON EXT SIGNAL	an interrupt made by an HP-UX system generated signal (BASIC/UX only; see chapter 18, “Interrupts and Timeouts.”)
ON HIL EXT	an interrupt generated by an HP HIL (Human Interface Link) device (see the <i>HP BASIC 6.2 Interface Reference</i> manual).
ON INTR	an interrupt generated by an an interface (see chapter 18, “Interrupts and Timeouts.”)
ON KBD	an interrupt generated by pressing a key (see the “Communicating with the Operator” chapter of the <i>HP BASIC 6.2 Advanced Programming Techniques</i> manual).
ON KEY	an interrupt generated by pressing a softkey: f1 thru f8 on ITF keyboards, or k0 through k9 on 98203 keyboards (see the “Communicating with the Operator” chapter of the <i>HP BASIC 6.2 Advanced Programming Techniques</i> manual).
ON KNOB	an interrupt generated by turning a knob—a rotary pulse generator: the built-in knob of a 98203 keyboard, an HIL knob, or one of the knobs on a Control Dial box (see the “Communicating with the Operator” chapter of the <i>HP BASIC 6.2 Advanced Programming Techniques</i> manual)
ON SIGNAL	an interrupt generated by a SIGNAL statement (see the <i>HP BASIC Language Reference</i>)
ON TIME	an interrupt from the clock when the clock reaches a specified time (see the “Clock and Timers” chapter of this manual)
ON TIMEOUT	an interrupt generated when an interface or device has taken longer than a specified time to respond to a data-transfer handshake (see chapter 18, “Interrupts and Timeouts.”)

2 Example of Event-Initiated Branching

The best way to understand how event-initiated branches operate in a program is to sit down at the computer and try a few examples. Start by entering the following short program, which sets up and enables branches when one of the softkeys is pressed.

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

Notice the various structures in this sample program. 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. (Disabling and deactivating are discussed later.)

The program segment labeled Spin is an infinite loop. If it weren't for interrupts, this program couldn't do anything except display a zero. However, there is an implied "if ... then" at the end of each BASIC program line due to the ON KEY action. This allows a selection process to occur. Either the Plus or the Minus subroutine can be selected as a result of softkey presses. These are normal subroutines terminated with a RETURN statement. (In the context of interrupt programming, these subroutines are called **service routines**.) The following section of "pseudo-code" shows what the program flow of the Spin segment actually looks like to the computer.

```
Spin: display X
  if Key0 then gosub Plus
  if Key5 then gosub Minus
  goto Spin
```

This pseudo-code is an over-simplification of what is actually happening, but it shows that the Spin segment is not really an infinite loop with no decision-making structure. Actually, most programs that use event-initiated

2-24 Program Structure and Flow

branching to control program flow will contain what appears to be an infinite loop. That is the easiest way to “keep the computer waiting” while it is waiting for an interrupt.

Now run the sample program you just entered. Notice that the bottom two lines of the screen display an inverse-video label area (this one is shown when using an ITF keyboard).

Inc

Dec

These labels are arranged to correspond to the layout of the softkeys. The labels are displayed when the softkeys are active and are not displayed when the softkeys are not active (See the “Communicating with the Operator” chapter of the *HP BASIC 6.2 Advanced Programming Techniques* manual for additional examples.) 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.

The starting value in the display line is zero, since numeric variables are initialized to zero at prerun. Each time you press **(k1)** or **(f1)**, the displayed value of X is incremented. Each time you press **(k5)** or **(f5)**, the displayed value of X is decremented. This simple demonstration should acquaint you with the basic action of the softkeys.

It is possible to make structures that are much more elaborate, with assignable priorities for each key, and keys that interrupt the service routines of other keys. There are many applications where priorities are not of any real significance, such as the example program running now. However, priorities will sometimes cause unexpected flow problems.

A full discussion on interrupt priority can be found in chapter 18, “Interrupts and Timeouts.” If you think you have an application that is “priority sensitive,” read that section carefully.

2 Example of Using the Knob

One characteristic of interrupt-driven program flow is that the computer's decisions can be more easily synchronized with the actions of devices connected to it. This type of application is often called **real-time programming**. An important example of real-time programming is machine control. A computer running an automatic packing machine must turn off the flow immediately when the jar is full. It is not acceptable for the computer to wait until the inventory printout is done and peanut butter is dumped all over the conveyor belt. Although machine control applications are very important, their extensive interfacing makes them inconvenient or impossible to use as demonstration programs in a manual such as this.

Another common example of real-time programming is computer games. The computer is expected to respond "instantly" to button presses, lever movement, etc. The operator expects immediate correlation between their input and the computer's output or display.

The following program is a very short example that demonstrates a real-time interaction between the knob (either the built-in knob on the HP 98203 keyboard or an HP-HIL knob) and the CRT. If you run this example program and turn the knob, you will see the kind of interaction that might be used for cursor control in a text editor. Obviously, a real cursor-control routine would be much more sophisticated, but this demonstrates the basic idea. (The "Communicating with the Operator" chapter of the *HP BASIC 6.2 Advanced Programming Techniques* manual also describes using the knob.)

```
10 ON KNOB .1 GOSUB Move_blip
20 Spin: GOTO Spin
30 !
40 Move_blip: !
50 PRINT TABXY(Spotx,Spoty);" ";
60 Spotx=Spotx+KNOBX/5
70 Spoty=Spoty+KNOBY/5
80 IF Spoty<1 THEN Spoty=1
90 IF Spoty>18 THEN Spoty=18
100 IF Spotx<1 THEN Spotx=1
110 IF Spotx>50 THEN Spotx=50
120 PRINT TABXY(Spotx,Spoty);CHR$(127);
130 RETURN
140 END
```

2-26 Program Structure and Flow

This example uses a short infinite loop to wait for pulses from the knob (line 20). Interrupts from the knob are enabled by the ON KNOB statement in line 10. The service routine erases the old “blip”, performs some scaling and range checking on the knob input, and prints the new “blip”.

Deactivating Events

Knowing how to “turn off” the interrupt mechanism is just as important as knowing how to enable it. Often, an event is a desired input during one part of the program, but not during another. You might use softkeys to set certain process parameters at the start of a program, but you don’t want interrupts from those keys once the process starts. For example, a report generating program could use a softkey to select single or double spacing. This key should be disabled once the printout starts so that an accidental key press does not cause the computer to abort the printout and return to the questions at the beginning of the program. On the other hand, you might want an “Abort” key that does precisely that. The important thing is that you decide on the desired action and make the computer obey your wishes.

Before going any further, let’s explain some important terminology. There are two general methods for “turning off” an interrupt. If an interrupt source is **deactivated**, it no longer has any influence on program flow. You can press a deactivated key all day long and nothing will happen. However, if an event is **disabled**, its action has only been temporarily postponed. The computer remembers that a key was pressed while it was disabled, and the action for that key will occur at the earliest opportunity once the disabled state is removed. There are examples in this section to demonstrate the difference between these two conditions.

All the “ON-event” statements have a corresponding “OFF-event” statement. This is one way to deactivate an interrupt source.

- OFF KEY deactivates interrupts from the softkeys. If a softkey is pressed while deactivated, it does nothing.
- OFF KNOB deactivates the ON KNOB interrupts. Turning the knob while ON KNOB is deactivated causes normal scrolling on the CRT.

The following example shows one use of OFF KEY to disable the softkeys. (Note that **(k1)** is used in the description. If you have an ITF keyboard, just substitute **(f1)**.) A softkey is used to select a parameter for a small printing

routine. Each press of **(k1)** increments and displays the step size that will be used as an interval between the printed numbers. When the desired step size has been selected, **(k4)** is pressed to start the printout. Enter and run this example. Notice that with line 240 and 250 commented out, the softkey menu, or label area, never changes.

```

100 Begin:  !
110  ON KEY 1 LABEL " DELTA" GOSUB Step_size
120  ON KEY 4 LABEL " START" GOTO Process
130  Inc=1
140  DISP "Step Size = 1"
150  !
160 Spin: GOTO Spin          ! Wait for key press
170  !
180 Step_size:  !
190  Inc=Inc+1              ! Change increment
200  DISP "Step Size =" ;Inc
210  RETURN
220  !
230 Process:  !
240 ! OFF KEY
250 ! ON KEY 8 LABEL " ABORT" GOTO Leave
260  Number=0
270  FOR I=1 TO 10
280    Number=Number+Inc
290    PRINT Number;
300    WAIT .6
310  NEXT I
320 Leave:  !
330  OFF KEY 8              ! Deactivate ABORT
340  PRINT                  ! End line
350  GOTO Begin            ! Start over
360  END

```

Now run the example again and press **(k1)** or **(k4)** while the printout is in progress. Notice that the keys are still active and produce undesired effects on the printing process. To “fix this bug,” remove the exclamation point from line 240. This disables all the softkeys when the printing process starts. Notice that the softkey menu goes away when no softkeys are active. This is a very handy feature while you are experimenting with interrupts. It provides immediate feedback to indicate when interrupts are active and when they are not.

Finally, remove the exclamation point from line 250. Now, the softkey menu appears during the printing process. However, the choices are different than

2-28 Program Structure and Flow

at the start of the program. The keys used to select the parameter and start the process are not active, because they are not needed at this point in the program. Instead, (k8) can be used to “gracefully” abort the process and return to the start of the program.

The OFF KEY statement can include a key number to deactivate a selected key. This was done in line 330.

Disabling Events

All the previous examples have shown complete deactivation of the softkeys. It is also possible to temporarily disable an event-initiated branch. This is done when an active event is desired in a process, but there is a special section of the program that you don’t want to be interrupted. Since it is impossible to predict when an external event will occur, the special section of code can be “protected” with a DISABLE statement. This is sometimes necessary to prevent a certain variable from being changed in the middle of a calculation or to insure that an interface polling sequence runs to completion. It is difficult in a short, simple example to show *why* you would need to do this. But it is not difficult to show *how* to do it.

```

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

```

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

Chaining Programs

With this BASIC system, it is also possible to “chain” programs together; that is, one program may be executed, which in turn loads and runs another, which in turn loads and runs yet another, and so forth. This section describes the available methods for chaining programs.

Using LOAD

The LOAD statement clears the current program, brings in a program from a PROG file, with the option of beginning program execution at a specified line. This type of LOAD is performed by adding a line identifier. For example, the following command tells the computer to load the program in file “STONE” and begin execution at line 10:

```
100 LOAD "STONE",10
```

The line identifier may be a label or a line number, but it must identify a line in the main program segment (not in a subprogram or user-defined function).

If you want to communicate any information to the program that is being loaded, you have the following two methods:

- Store the information in a file which both programs can access. (File access is fully explained in the “Data Storage and Retrieval” chapter.)
- Store the information in “common” (COM) variables which both programs can access. (Note that the programs must have *identical* COM declarations. COM is fully discussed in the “Subprograms and User-Defined Functions” chapter; a simple example is provided in the subsequent section describing how to use GET to chain programs.)

The LOAD command cannot be used to bring in arbitrary program segments or append to a main program like GET can.

Using GET

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

The following statement:

2-30 Program Structure and Flow

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

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

GET can also specify that program execution is to begin. This is done by specifying two line identifiers. For example:

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

specifies that:

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

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

Example of Chaining with GET

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

First Program Segment:

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



Program Segment in File Named "Wattage":

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

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

Program-to-Program Communications

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

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

If the program segments did not contain matching COM declarations, all variables in the mis-matched COM statements would be destroyed by the "pre-run" that the system performs after appending the new lines but before running the first program line. (For a definition of pre-run, see the "Subprograms and User-Defined Functions" chapter of this manual.)

Numeric Computation

Numeric computations deal exclusively with numeric values. Thus, adding two numbers and finding a sine or a logarithm are all numeric operations; while converting bases and converting a number to a string or a string to a number are not. (Converting bases and converting numbers to strings and strings to numbers are covered in the chapter on “String Manipulation.”)

Numeric Data Types

There are three numeric data types available in BASIC: INTEGER, REAL and COMPLEX. Any numeric variable that is not declared an INTEGER or COMPLEX variable is a REAL variable. This section covers these data types.

REAL Data Type

The valid range for REAL numbers is approximately:

$-1.797\ 693\ 134\ 862\ 315 \times 10^{308}$ thru $1.797\ 693\ 134\ 862\ 315 \times 10^{308}$

or

`-MAXREAL` thru `+MAXREAL`

which are the functions used to obtain the above range values.

The smallest non-zero REAL value allowed is approximately:

$\pm 2.225\ 073\ 858\ 507\ 202 \times 10^{-308}$

or

`± MINREAL`

A REAL can also have the value of zero.

INTEGER Data Type

An INTEGER can have any whole-number value from:

-32 768 thru +32 767

3 COMPLEX Data Type

A complex number is an ordered pair (x,y) denoted by Mathematicians as:

$$x + iy$$

where:

x is the real part of the complex number

y is the imaginary part of the complex number. The i in front of the y forms the imaginary number iy and is the same as multiplying y by the $\sqrt{-1}$. For example, the $\sqrt{-9}$ could be written as: $\sqrt{-1} \times \sqrt{9}$ or $3i$.

BASIC complex numbers are stored as two REAL numbers. This means that a complex number requires 16 bytes of memory (each REAL component takes 8 bytes).

REAL, INTEGER, and COMPLEX variables may be declared as arrays.

Numeric Variables

One of the most important principles of BASIC programming is the use of variables to keep track of data. Numeric variables are used for numeric data and string variables are used for string data. The use of numeric variables is covered in this section, while the use of string variables is covered in chapter 5, "String Manipulation."

3-2 Numeric Computation

Naming Variables

The first thing that you must do before assigning a variable is to choose a name for it. A numeric variable name must conform to the following rules:

1. The maximum length is 15 characters.
2. The first character must be a capital letter.
3. All other characters in the name must be either lower-case letters, numerals, or the underscore (.). These characters can be arranged in any order.

For example, `Income_1991` is a valid variable name, but `Income-1991` is not. `A1` is valid, but `1A` is not. `Location_1111_a` is valid, but `Location_1111_A` is not.

String variable names follow the same rules, but must have a trailing dollar sign (\$) appended. Up to 15 characters, *plus* the dollar sign, are allowed.

Declaring Variables

It is good programming practice to declare the data type of all variables used in a program. The `INTEGER`, `REAL`, and `COMPLEX` statements have been provided to accomplish this task. However, `BASIC` is forgiving and implicitly assumes a variable is `REAL` if its type is not explicitly declared. Here are some examples of explicitly declaring variables:

```
INTEGER I, J, Days(5), Weeks(5:17)
REAL X, Y, Voltage(4), Hours(5,8:13)
COMPLEX S, T, Phasor_1(10), Phasor_2(10)
```

Each of the above statements declares two scalar and two array variables. A scalar is a variable which can, at any given time, represent a single value. An array is a subscripted variable that may contain multiple values accessed by subscripts. It is possible to specify both the lower and upper bounds of an array or to specify the upper bound only, and use the existing `OPTION BASE` as the lower bound. Details on declarations of arrays and how to use them are provided later in this chapter when arrays are dealt with in detail. The `DIM` statement may also be used to declare a `REAL` array.

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

An `ALLOCATE` statement can be used to declare `REAL`, `INTEGER`, and `COMPLEX` arrays. The `ALLOCATE` and `DEALLOCATE` statements allow

you to dynamically allocate memory in programs which need tight control over memory usage.

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

DEALLOCATE Co_ords(*), Status(*)
DEALLOCATE Poles(*), Location(*)
```

3

Assigning Variables

The following statements are equivalent:

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

To assign values to COMPLEX variables, the variables must first be declared as COMPLEX.

```
10 COMPLEX B,C,D
20 B=3.0          ! Real part = 3.0; imaginary part = 0.0.
30 C=CMPLX(3,4) ! Creates a COMPLEX value and assigns it to C.
40 D=CMPLX(Real_part,Imaginary_part)
50 B=D ! Assigns both the real and imaginary parts of D to B.
60 .
70 .
```

Implicit Type Conversions

The computer will automatically convert between REAL, INTEGER, and COMPLEX values in assignment statements and when parameters are passed by value in function and subprogram calls. The type conversion rules are:

- When a value is assigned to a variable, the value is converted to the data type of that variable.

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

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

3-4 Numeric Computation

Executing this program returns the following result:

```
2.34      2
```

INTEGER and REAL data types are converted to COMPLEX data types by adding an imaginary part of 0.

```
100 COMPLEX Complex_var1, Complex_var2
110 REAL Real_var
120 INTEGER Integer_var
130 Real_var=1.22
140 Integer_var=4
150 Complex_var1=Real_var
160 Complex_var2=Integer_var
170 DISP Complex_var1, Complex_var2
180 END
```

3

Executing this program produces the following results:

```
1.22 0      4 0
```

COMPLEX data types are converted to INTEGER and REAL data types by dropping the imaginary part.

```
100 COMPLEX Complex_var
110 REAL Real_var
120 INTEGER Integer_var
130 Complex_var=CMPLX(1.22,4.11)
140 Real_var=Complex_var
150 Integer_var=Complex_var
160 DISP Real_var, Integer_var
170 END
```

Executing this program produces the following results:

```
1.22      1
```

- Conversions that occur within expression convert to the “highest” or most complicated data type before the operation occurs. For example:

```
CMPLX(3,-1) + 4.56
```

converts the REAL data type 4.56 to a COMPLEX value before the addition operation is performed.

When parameters are passed by value, the type conversion is from the data type of the calling statement’s parameter to the data type of the subprogram’s parameter. This type conversion occurs automatically. When parameters are

passed by reference, the type conversion is not made and a type mismatch error will be reported if the calling parameter and the subprogram parameters are of different types.

Whenever numbers are converted from REAL to INTEGER representations, information can be lost. There are two potential problem areas in this conversion: rounding errors and range errors.

3

BASIC will automatically convert between types when an assignment is made. This presents no problem when an INTEGER is converted to a REAL. However, when a REAL is converted to an INTEGER, the REAL is rounded to the closest INTEGER value. When this is done, all information about the number to the right of the radix (decimal point) is lost. If the fractional information is truly not needed, there is no problem, but converting back to a REAL will not reconstruct the lost information—it stays lost.

Another potential problem with REAL to INTEGER conversions is the difference in ranges. While REAL values range from approximately -10^{308} to $+10^{308}$, the INTEGER range is only from $-32\,768$ to $+32\,767$ (approximately -10^4 thru $+10^4$). Obviously, not all REAL values can be rounded into an equivalent INTEGER value. This problem can generate INTEGER Overflow errors.

While the rounding problem is important, it does not generate an execution error. The range problem *can* generate an execution error, and you should protect yourself from crashing the program by either testing values before assignments are made, or by using ON ERROR to trap the error, and making corrections after the fact.

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

```
200 IF X > 32767 THEN X= 32767
210 IF X < -32768 THEN X = -32768
220 Intx = X
```

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

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

Both these methods avoid the overflow errors, but lose the fact that the values were originally out of range. If out-of-range is a meaningful condition, an error handling trap is more appropriate.

3-6 Numeric Computation

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

Precision and Accuracy: The Machine Limits

3

Your computer stores all REAL variables with a sign, approximately 15 significant digits, and the exponent value. For most engineering and other applications, rounding errors are not a problem because the resolution of the computer is well beyond the limitations of most scientific measuring devices. However, when high-resolution numerical analysis requires accuracy approaching the limits of the computer, round-off errors must be considered.

Rounding errors should be considered when conversions are made between decimal digits and binary form (the form used by the computer internally to store the values). Input and output operations are occasions when this can occur. Given the format used for REALs, the conversion REAL \rightarrow decimal \rightarrow REAL will yield an identity only if the REAL \rightarrow decimal conversion produces a 17-decimal-digit mantissa and the calculations for the conversions are done in extra precision. This is not the case on HP Series 200/300 BASIC. Therefore, several things can be said about these conversions on HP Series 200/300 BASIC:

- Up to and including 16 decimal digits are allowed when storing a number in internal form. If there are more digits, they are ignored.
- Up to and including 15 decimal digits may be output when converting a REAL for printing, display, etc.. A full 16-digit conversion is not allowed because there are not 16 full digits of precision.
- It is possible for two distinct decimal numbers to map onto the same REAL number because the binary mantissa does not have enough bits to represent all 16 decimal digits. This can happen only if the decimal numbers are specified to 16-digits.
- It is possible for two distinct REAL numbers to convert to the same decimal number even if the conversion is done to 15-decimal-digit accuracy. Therefore, you cannot use a comparison of the digits in printed or displayed numbers to check for equality.

3

- All distinct 15 digit decimal strings have a correct distinct REAL representation, but it is not always possible to map them onto their correct representation because REAL multiplies are not done in extra precision, and the table entries are only 64 bits. In other words, the decimal → REAL conversion may produce a REAL that differs from the true representation by a maximum of two bits.

There are references at the end of this chapter to documents that contain further information on the subject of representing real numbers.

Note that when incrementing a value repeatedly by a fraction, the value may become inexact in the last decimal places. For example, the following program will print some values for `FRACT(I)` that are not divisible by `.1`. This is an unavoidable consequence of the efficient binary notation used in the computer.

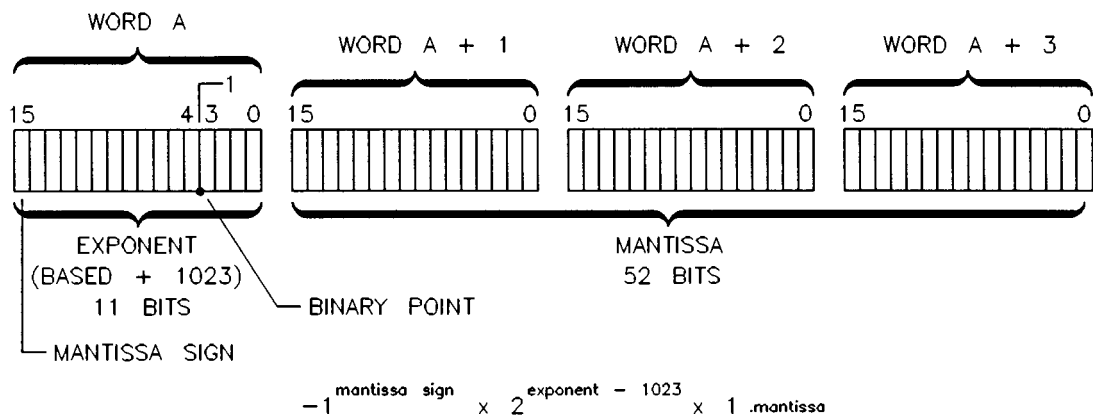
```

REAL I
I=0
LOOP
  I=I+.1
  PRINT I,FRACT(I)
END LOOP

```

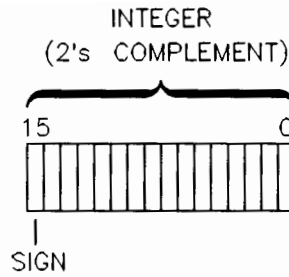
Internal Numeric Formats

The storage format for REAL and INTEGER numbers in memory are as follows:



Storage Format for REAL Variables

3-8 Numeric Computation



Storage Format for INTEGER Variables

Note that COMPLEX values are stored as 2 REAL numbers with the real part first and the imaginary part following.

Evaluating Scalar Expressions

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

- Hierarchy of expression evaluation
- Delayed assignment surprise
- BASIC operators: monadic, dyadic, and relational

The Hierarchy

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

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

Computers do not deal well with ambiguity, so a hierarchy is used for evaluating expressions to eliminate any questions about the meaning of an expression. Six items can appear in a numeric expression:

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

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

Math Hierarchy

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

When an expression is being evaluated it is read from left to right and operations are performed as encountered, unless a higher precedence operation

3-10 Numeric Computation

is found immediately to the right of the operation encountered, or unless the hierarchy is modified by parentheses.

The Delayed Assignment Surprise

BASIC delays assigning a value to a variable as long as possible. This means that when a variable is in an area of COM accessible to both the main program and a user-defined function is used in an expression that also calls the user-defined function—and is modified in the function—the value of the expression can be surprising, although not unpredictable. For example, if we define a function `FNNeg1` that returns a minus 1, we would expect the following lines to print 2.

```
10 COM X
20 X = 3
30 Y = X + FNNeg1
40 PRINT Y
```

However, if the user-defined function looks like this:

```
1000 DEF FNNeg1
2000     COM X
1020     X = 500
1030     RETURN -1
1040 FNEND
```

The actual result will be 499. Therefore, don't use a user-defined function to modify values of variables. They are designed for returning a single value, and are best reserved for that.

Operators

There are three types of operators in BASIC:

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

Note

The *only* relational operators allowed with COMPLEX values are: = and <>. The *only* dyadic operators allowed with COMPLEX values are: ^, +, -, *, /, <>, and =. The *only* monadic operators allowed with COMPLEX values are: + and -.

3

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

Expressions as Pass Parameters

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

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

Comparisons Between Two REAL or COMPLEX Values

If you are comparing INTEGER numbers, no special precautions are necessary since these values are represented *exactly*. However, if you are comparing REAL or COMPLEX values, especially those which are the results of calculations and functions, it is possible to run into problems due to rounding and other limits inherent in the system. For example, consider the use of relational operators in IF..THEN statements to check for equality in any situation resembling the following:

```
1220  DEG
1230  A=25.3765477
1240  IF SIN(A)^2+COS(A)^2=1 THEN
1250    PRINT "Equal"
1260  ELSE
1270    PRINT "Not Equal"
1280  END IF
```

You may find that the equality test fails due to rounding errors or other errors caused by the inherent limitations of computers. For additional information on rounding errors read the subsequent section titled "Rounding Functions."

3-12 Numeric Computation

Numerical Functions

Intrinsic functions are built into BASIC. This section discusses various intrinsic numerical functions in the following categories. For a list of all the numerical functions, see the “Keyword Summary” in the *HP BASIC Language Reference* and *HP BASIC Condensed Reference*.

3

Arithmetic Functions

It is not always convenient to write a program segment or subprogram to perform these numeric operations. To eliminate this inconveniences, BASIC provides you with the following 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	Return the square root of an expression. Takes a REAL, INTEGER, or COMPLEX number as their 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.

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 the elements in a numeric array.

For more information on numeric array functions, read the “Numeric Arrays” chapter.

Exponential Functions

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

EXP	Raise the Napierian e to a power. $e \approx 2.718\ 281\ 828\ 459\ 05$.
LGT	Returns the base 10 logarithm of an expression.
LOG	Returns the natural logarithm (Napierian base e) of an expression.

Trigonometric Functions

Six trigonometric functions and the constant π are provided for dealing with angles and angular measure. Note that all trigonometric functions take REAL, 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 an angle.
SIN	Returns the sine of an angle.
TAN	Returns the tangent of an angle.
PI	Returns an approximate value for π .

Trigonometric Modes: Degrees and Radians

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

Hyperbolic Functions

Six hyperbolic functions are available with the BASIC system 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

BASIC provides these functions to deal with binary numbers:

BINAND	Returns the bit-by-bit logical and of two numbers.
BINCMP	Returns the bit-by-bit complement of its numbers.
BINEOR	Returns the bit-by-bit exclusive or of two numbers.
BINIOR	Returns the bit-by-bit inclusive or of two numbers.
BIT	Returns the value (0, 1) of a specified bit of a number.
ROTATE	Returns the value obtained by shifting a number a specific number of bit positions, <i>with</i> wraparound.
SHIFT	Returns the value obtained by shifting a number a specific number of bit positions, <i>without</i> wraparound.

When any of these functions are used, the arguments are first converted to **INTEGER** (if they are not already in the correct form) and then the specified operation is performed. It is best to restrict bit-oriented binary operations to declared **INTEGER**s.

Limit Functions

It is sometimes necessary to limit the range of values of a variable.

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

These functions work with **INTEGER** and **REAL** values.

Rounding Functions

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

DROUND	Rounds a numeric expression to the specified number of digits. If the specified number of digits is greater than 15, no rounding takes place. If the number of digits specified is less than 1, zero is returned.
PROUND	Returns the value of the argument rounded to a specified power of ten.

Rounding Errors Resulting from Comparisons

Equality errors occur when multiplying or dividing data values and comparing their result to another non-integer data value. This happens because the 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, here are three methods that can be used to eliminate equality errors which could occur as a result of this:

- Use the **DROUND** function to eliminate unwanted resolution *before* comparing results.
- Use the absolute value of the difference between the two values, and test for the difference less than a specified limit.
- Use the absolute value of the relative difference between two values, and test for the difference less than a specified limit:

```
IF ABS((C-F)/C) < 10^(-Delta_power) THEN PRINT "C is equal to F"
```

The following example shows the **DROUND** technique:

```
1050  A=32.5087
1060  B=31.625
1070  C=A*B          ! Product is 1028.08763750
1080  D=32.5122
1090  E=31.621595509
1100  F=D*E          ! Product is 1028.08763751
1110  IF C=F THEN 1130
1120  PRINT "C is not equal to F"
1130  C=DROUND(C,7)
1140  F=DROUND(F,7)
1150  IF C=F THEN
1160    PRINT "C equals F after DROUND"
```

```

1170 ELSE
1180 PRINT "C not equal to F after DROUND"
1190 END IF
1200 END

```

3

Here is an example of the absolute value method of testing equality. In this case, a difference of less than 0.001 is assumed to be evidence of adequate equality. Using the previous example, we change methods starting at line 1130.

```

1130 IF ABS(C-F)<.001 THEN
1140 PRINT "C is equal to F within 0.001"
1150 ELSE
1160 PRINT "C is not equal to F within 0.001"
1170 END IF
1180 END

```

Finally, here is an example of the relative difference method. Once again, we change methods starting at line 1130.

```

1130 IF ABS((C-F)/C)< 10-3 THEN
1140 PRINT "Relative difference between C and F less than 10-3"
1150 ELSE
1160 PRINT "Relative difference between C and F greater than 10-3"
1170 END IF
1180 END

```

Random Number Function

BASIC provides a pseudorandom number generator:

RND Returns a pseudo-random number greater than 0 and less than 1.

Since many modeling systems require random numbers with arbitrary ranges, it is necessary to scale the numbers.

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

Note that the above statement will return an integer between *Offset* and *Offset + Range*.

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

3-18 Numeric Computation

Complex Functions

These functions are available when the COMPLEX binary is loaded. Topics which are covered in this section are:

- Creating COMPLEX Values
- Evaluating COMPLEX Numbers
- COMPLEX Arguments and the Trigonometric Mode
- Determining the Parts of COMPLEX Numbers
- Converting from Rectangular to Polar Coordinates
- An Application for COMPLEX Numbers

Creating COMPLEX Values

The CMPLX function lets you create a complex value from a pair of REALS.

```
10 COMPLEX B,C
20 C=CMPLX(3.5,.5)
30 B=C
40 PRINT C,B
50 END
```

Executing the above program produces these results:

```
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, executing this statement:

```
SQRT(-1)
```

results in this error:

```
ERROR 30
```

The square root of a COMPLEX value whose real part is negative is defined so the operation is allowed. For example, executing this statement:

```
SQRT(CMPLX(-1,0))
```

returns the value:

```
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 or RAD). 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:

```
.5           degree mode
-.988031624093  0  radian mode
.5           degree mode
```

Note



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

Determining the Parts of COMPLEX Numbers

In many cases, 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.

3-20 Numeric Computation

REAL(C) returns the real part of a complex number.
IMAG(C) returns the imaginary part of a complex number.
CONJG(C) returns the complex conjugate of a complex number. That is:

`CONJG(CMPLX(3,4))`

is the same as

`CMPLX(3,-4)`

For example, executing the following statement:

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

produces this result:

```
10
```

This next statement:

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

produces:

```
-3
```

This last example:

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

produces:

```
10 3
```

Converting from Rectangular to Polar Coordinates

BASIC stores and uses complex numbers in a representation called rectangular coordinates. The rectangular coordinate representation of the complex plane is a Cartesian coordinate system where 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 representation for polar coordinates is given as follows:

$$M \angle \theta$$

where M is the magnitude and θ is the argument. The BASIC function used to obtain the magnitude is `ABS`, and the function used to obtain the argument is `ARG`.

The following program converts the rectangular coordinates 5 and 6 of the complex number $5 + i6$ to polar coordinates.

3

```
140 RAD
150 DISP "The magnitude of 5 + i6 is: ";ABS(CMPLX(5,6))
160 DISP "The argument of 5 + i6 is: ";ARG(CMPLX(5,6))
170 END
```

Executing this program produces the following:

```
The magnitude of 5 + i6 is: 7.81024967591      in RAD mode
The argument of 5 + i6 is: .876058050598
```

Time and Date Functions

The following functions return the time and date in seconds:

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

(If there is no battery-backed clock, the clock value set at power-on is 2.086 629 12E+11, which represents midnight March 1, 1900. If the computer is connected to an SRM system, the SRM clock's value is read from the SRM System's clock when the SRM binary is loaded.)

The time value accumulates from its initial value unless it is changed by `SET TIME` or `SET TIMEDATE`. For example, executing this function

```
TIMEDATE
```

returns a value in seconds similar to the following:

```
2.11404868285E+11
```

TIME converts a formatted time-of-day string into a numeric value of seconds passed midnight. For example, executing this statement:

```
TIME("8:37:30")
```

returns the following numeric value in seconds:

```
31050
```

3-22 Numeric Computation

DATE converts a formatted date string into a numeric value in seconds. For example, executing this statement:

```
DATE("26 OCT 1986")
```

returns the following numeric value in seconds:

```
2.11397472E+11
```

For more information on this subject read the chapter in this manual titled "The Real-Time Clock." Also included in this chapter are the DATE\$ and TIME\$ string functions.

Base Conversion Functions

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

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

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

returns the following numeric value:

```
5600
```

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

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

returns the following numeric value:

```
-4
```

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

General Functions

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

```
ENTER 2;Numeric_value
```

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

```
ENTER KBD;Numeric_value
```

where you know the function KBD must stand for keyboard. Therefore, you know the statement is asking you to enter a numeric value from the keyboard.

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

CRT	Returns the INTEGER 1. This is the select code of the internal CRT.
KBD	Returns the INTEGER 2. This is the select code of the keyboard.
PRT	Returns the INTEGER 701. This is the default (factory set) device selector for an external printer (connected through the built-in HP-IB interface at select code 7).
SC	Returns the interface select code associated with an I/O path name.

Another function which fits in the general function category is the RES function. This function returns the last live keyboard numeric result (same as **RECALL** key).

Floating-Point Math

HP BASIC supports various floating-point coprocessors to provide faster computation.

- The HP 98635A Floating-Point Math Card is supported for Series 200 and Models 310 and 320 (BASIC/WS only).
- The MC68881/MC68882 Floating-Point Coprocessor is supported for Models 330, 332, 340, 345, 350, 360, 370, and 375 (for both BASIC/WS and BASIC/UX).
- The MCMATH binary provides MC68881/MC68882 compatible floating-point math routines for Models 380 and 382 (for both BASIC/WS and BASIC/UX).
- The HP 82327 Floating-Point Unit (an MC68882) is supported for the HP 82324 High-Performance Measurement Coprocessor (BASIC/DOS).

For a complete description of floating-point math hardware support, refer to *HP BASIC 6.2 Advanced Programming Techniques*.

Numeric Arrays

An array is a multi-dimensional structure of variables that are given a common name. The array can have one to six dimensions. Each location in an array contains one value, and each value has the characteristics of a single variable, either **REAL**, **INTEGER** or **COMPLEX** (string arrays are discussed in the chapter, “String Manipulation”). Note that many of the statements that deal with arrays require the **MAT** binary.

A one-dimensional array consists of n elements, each identified by a single subscript. A two-dimensional array consists of m times n elements where m and n are the maximum number of elements in the two respective dimensions. Arrays require a subscript in each dimension, in order to locate a given element of the array. Arrays are limited to six dimensions, and the subscript range for each dimension must lie between -32767 and 32767. **REAL** arrays require eight bytes of memory for each element, plus overhead, 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

For more information about numeric arrays, array operations, matrices, and vectors, refer to *HP BASIC 6.2 Advanced Programming Techniques*.

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)
```

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

When you dimension an array, the system reserves space in internal memory for it. The system also sets up a table which it uses to locate each element in the array. The location of each element is designated by a unique combination of subscripts, one subscript for each dimension. For a two-dimensional array, for instance, each element is identified by two subscript values. An example of dimensioning a two-dimensional array is as follows:

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

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

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

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

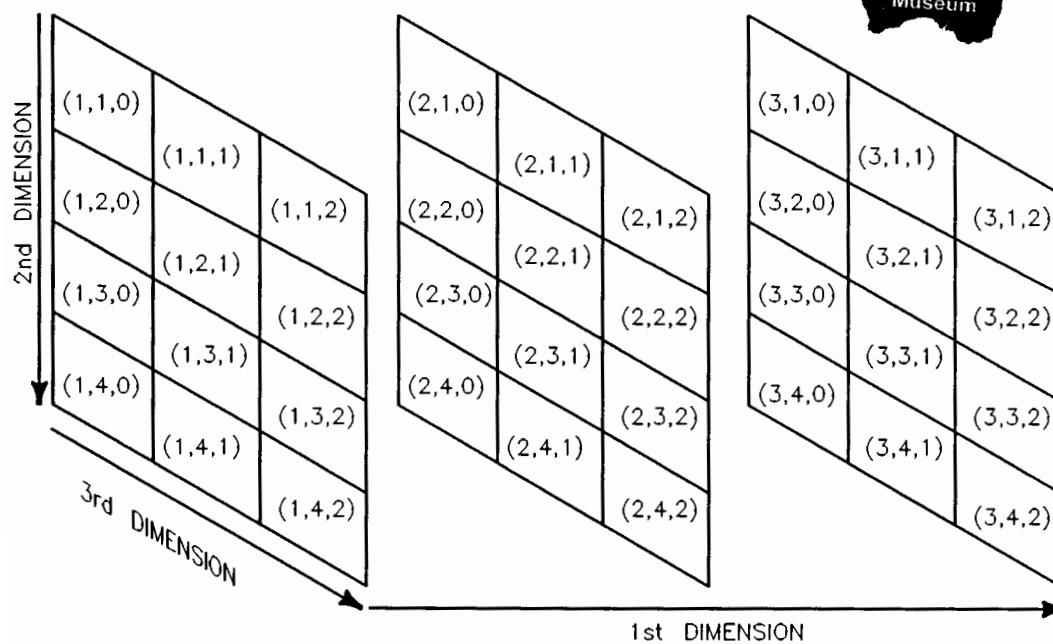
4-2 Numeric Arrays

Some Examples of Arrays

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

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

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



Planes of a Three-Dimensional REAL Array



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

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

4

Two-Dimensional REAL ARRAY

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

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

4-4 Numeric Arrays

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

A Dynamically Allocated, Two-Dimensional INTEGER Array

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

4

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

Note



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

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

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

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

Problems with Implicit Dimensioning

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

4

Finding Out the Dimensions of an Array

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

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

would print 3.

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

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

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

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

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

would return a 0, while,

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

would return a -1.

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

4-6 Numeric Arrays

These functions are powerful tools for writing programs that perform functions on an array regardless of the array's size or shape.

Using Individual Array Elements

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

Assigning an Individual Array Element

Initially, every element in an array equals zero. There are a number of different ways to change these values. The most obvious is to assign a particular value to each element. This is done by specifying the element's subscripts.

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

Extracting Single Values From Arrays

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

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

BASIC automatically converts variable types. For example, if you assign an element from a COMPLEX array to an INTEGER variable, the system will round the real part and ignore the imaginary part of the COMPLEX number.

Filling Arrays

This section discusses three methods for filling an entire array:

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

Assigning Every Element in an Array the Same Value

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

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

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

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.

Using the READ Statement to Fill an Entire Array

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

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

The asterisk in line 140 is used to designate the entire array rather than a single element. Note also that the right-most subscript varies fastest. In this case, it means that the system fills an entire row before going to the next one. The READ/DATA statements are discussed further in the chapter “Data Storage and Retrieval”.

4-8 Numeric Arrays

Executing the previous program produces the following results:

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

Copying Entire Arrays into Other Arrays

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

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

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

```
MAT A= B
```

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

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

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

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

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

We can execute,

4

```
MAT A= B
```

since **A** is dimensioned to 9 elements and **B** is only 4 elements. The copy automatically redimensions **A** to a 2x2 array. Nevertheless, we can still execute:

```
MAT A= C
```

This works because the nine elements originally reserved for **A** remain available until the program is scratched. **A** now becomes a 2x4 matrix. After **MAT A= C**, we could not execute:

```
MAT B= A
```

or

```
MAT B= C
```

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

```
MAT C= A
```

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

4-10 Numeric Arrays

Printing Arrays

Printing an Entire Array

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

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

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

4

Examples of Formatting Arrays for Display

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

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

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

Assuming that you intended to display a 5x5 array, your results should look similar to this:

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

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

4

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

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

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

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

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

Plane 2
3   3   3
3   3   3
3   3   3
```

4-12 Numeric Arrays

```

Plane 3
3   3   3
3   3   3
3   3   3

```

Passing Entire Arrays

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

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

4

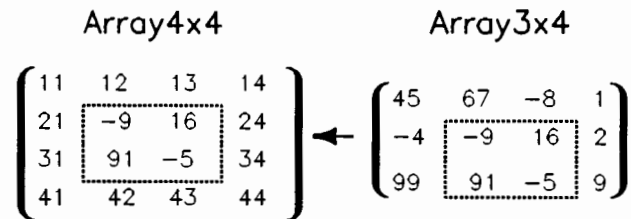
Copying Subarrays

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

```
MAT Array55= Array33
```

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

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



Copying a Subarray into Another Subarray

Topics discussed in this section are:

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

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

Subarray Specifier

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

`Array_name(subarray_specifier)`

`String_array$(subarray_specifier)`

The above subarray could take on many “sizes” and “shapes” depending on what you used as dimensions for the array and the values used in the *subarray_specifier*. Note that “size” refers to the number of elements in the subarray and “shape” refers to the number of dimensions and elements in each dimension, respectively [e.g. both of these subscript specifiers have the same shape: (-2:1, -1:10) and (1:4, 9:20)]. Before looking at ways you can express a subarray, let’s learn a few terms related to the subarray specifier.

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

subscript expression is an expression which reduces the **RANK** of the subarray. For example if you wanted to select a one-element subarray from a two-dimensional array which is located in the 2nd row and 3rd

4-14 Numeric Arrays

column, you would use the following subarray specifier: (2,3:3). The subscript expression in this subarray specifier is 2 which restricts the subarray to row 2 of the array.

default range

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

4

Some examples of subarray specifiers are as follows:

- (1,*) a subscript expression and a default range which designate the first row of a two-dimensional array.
- (1:2) a given subscript range which represents the first two elements of a one-dimensional array.
- (*,-1:2) a default range and subscript range which represents all of the elements in the first four columns of a two-dimensional array (base of 2nd dimension assumed to be -1).
- (3,1:2) a subscript expression and subscript range which represent the first two elements in the third row of a two-dimensional array.
- (1,*,*) a subscript expression and two default ranges which represent a plane consisting of all the rows and columns of the first plane in the first-dimension.
- (1,1:2,*) a subscript expression, subscript range and default range which represent the first two rows in the first plane of the first-dimension.
- (1,2,*) two subscript expressions and a default range which represent the entire second row in the first plane of the first-dimension.

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

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

Copying an Array into a Subarray

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

4

A destination and source array are dimensioned as follows:

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

Suppose these arrays contain the following INTEGER values:

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

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

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

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

`Des_array`

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

4

Copying a Subarray into an Array

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

A destination and source array are dimensioned as follows:

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

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

<code>Des_array</code>	<code>Sor_array</code>
<code>(-1 14 8 4 98 43 90 -3)</code>	<code>(-11 -4 1 2 3 4 78 100 8 18)</code>

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

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

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

Des_array
 (-4 1 2 3 4 78)

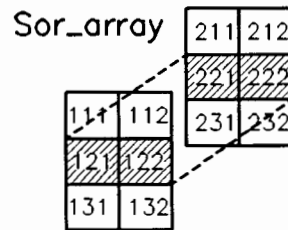
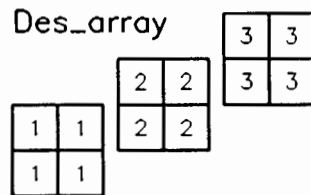
Copying a Subarray into another Subarray

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

A destination and source array are dimensioned as follows:

4
 100 OPTION BASE 1
 110 DIM Des_array(3,2,2),Sor_array(2,3,2)
 120 .
 130 .

Suppose these three-dimensional arrays contain the following values:

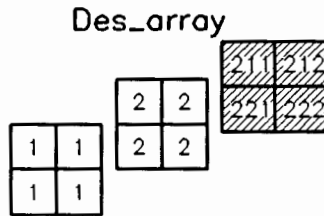


in order to properly copy a source subarray ($Sor_array(*,2,*)$) into a destination subarray using asterisks to represent the ranges of dimensions, you would use line 190 given below:

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

4-18 Numeric Arrays

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



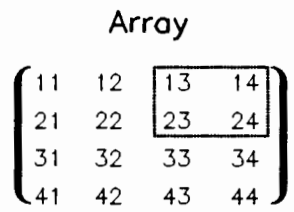
Copying a Portion of an Array into Itself

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

A destination and source array are dimensioned as follows:

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

Suppose this two dimensional array contains the following values:



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

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

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

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

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

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

4 will always generate a run-time error.

Rules for Copying Subarrays

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

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

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

is the correct way of copying a subarray into another subarray provided the default range given in the source array (Sor_array) has only two elements in it. Note that the source array is a three-dimensional array. However, it still meets the criteria of having the same number of ranges as the destination array because two of its entries are ranges and one is an expression.

4-20 Numeric Arrays

- If two subarrays are given in a **MAT** statement, the subscript ranges in the source array *must be* the same shape as the subscript ranges in the destination array. For example, the following example is *legal*:

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

however, the following example is *not legal*:

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

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

Redimensioning Arrays

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

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

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

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

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

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

The new array now looks like the figure below:

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

Note that it retains the values of the elements, though not necessarily in the same locations. For instance, **A**(2,1) in the original array was 4, whereas in the

redimensioned array it equals 5. For example, if we REDIMed A again, this time to a 2x2 array, we would get:

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

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

We could then initialize all elements to 0:

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

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

4

It is also important to realize that elements that are out of range in the REDIMed array still retain their values. The fifth through ninth elements in A still equal 5 through 9 even though they are now inaccessible. If we REDIM A back to a 3x3 array, these values will reappear. For example:

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

results in:

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

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

```
10 OPTION BASE 1
20 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.

4-22 Numeric Arrays

String Manipulation

A word, a name or a message can be stored in the computer as a **string**. Any sequence of characters may be used in a string. Quotation marks are used to delimit the beginning and ending of the string. For example:

```
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. (Refer to "Naming Variables" in chapter 3 for more information.)

5

Details About Strings

If you are using a localized version of BASIC that supports two-byte characters, such as Japanese localized BASIC, note that BASIC handles two-byte characters in special ways. This chapter describes the character handling features of BASIC *as they apply to one-byte extended ASCII characters*.

In general, BASIC handles one- and two-byte characters in a very similar manner; however, two-byte characters are handled differently in some cases. The general principles of two-byte character handling are explained in *HP BASIC 6.2 Porting and Globalization*.

The **length** of a string is the number of characters in the string. In the previous example, the length of A\$ is 8 since there are eight characters in the literal

"COMPUTER". A string with length 0 (i.e., that contains no characters) is a null string. 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.

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
```

5

Result: The time is "NOW".

String Storage

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

DIM Long\$[400]	<i>Reserve space for a 400-character string</i>
COM Line\$[80]	<i>Reserve an 80-character common variable</i>
ALLOCATE Search\$[Length]	<i>Dynamic length allocation</i>

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

5-2 String Manipulation

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 disk.

5

Evaluating Expressions Containing Strings

This section covers:

- Evaluation hierarchy
- String concatenation
- Relational operations



Evaluation Hierarchy

The three allowed string operations are extracting a substring, concatenation, and parenthesization. The evaluation hierarchy is presented in the following table.

Order	Operation
High	Parentheses
—	Substrings and Functions
Low	Concatenation

String Concatenation

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

```

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

```

5

Result:

```

WRIST    WATCH    WRISTWATCH

```

Relational Operations

Most of the relational operators used with numeric expressions can also be used with strings. Any of these relational operators may be used: <, >, <=, >=, =, <>.

The following examples show some of the possible tests.

```

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

```

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

5-4 String Manipulation

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

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

String\$[4] Specifies a substring starting with the fourth character of the original string.

The subscript must be in the range: 1 to the current length of the string plus 1. Note that the brackets now indicate the substring's starting position instead of the total length of the string as when reserving storage for a string.

Subscripted strings may appear on either side of the assignment.

Single-Subscript Substrings

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

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

Statement	Output
PRINT A\$	DICTIONARY
PRINT A\$[0]	(error)
PRINT A\$[1]	DICTIONARY
PRINT A\$[5]	IONARY
PRINT A\$[5,2]	
PRINT A\$[10]	Y
PRINT A\$[11]	(null string)
PRINT A\$[12]	(error)

5

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

Double-Subscript Substrings

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

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

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

5-6 String Manipulation

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

5

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

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

Special Considerations

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

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

Result: **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. When a substring assignment specifies fewer characters than are available, any extra trailing characters are truncated. For example:

```
10 Big$="Too big to fit"
20 Small$="Little string"
30 !
40 Small$[1,3]=Big$
50 !
60 PRINT Small$
70 END
```

5 Result: Tootle string

The alternate assignment is shown in the next example. Here a 4-character string is assigned to a 8-character substring.

```
10 Big$="A large string"
20 Small$="tiny"
30 !
40 Big$[3,10]=Small$
50 !
60 DISP Big$
70 END
```

Prints: A tiny ring

Since the subscripted length of the substring is greater than the length of the replacement string, enough blanks (ASCII spaces) are added to the end of the replacement string to fill the entire specified substring.

5-8 String Manipulation

String-Related Functions

Several built-in functions are available in BASIC for manipulating strings. These functions include conversions between string and numeric values.

Special Note for Localized BASIC

If you are using a localized version of BASIC that supports two-byte characters, such as Japanese localized BASIC, note that BASIC handles two-byte characters in special ways. The general principles of two-byte character handling are explained in *HP BASIC 6.2 Porting and Globalization*.

Current String Length

The “length” of a string is the number of characters in the string. The `LEN` function returns an integer equal to the string length. For example:

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

Result: 7

The following example program prints the length of a string that is typed on the keyboard.

```
10 DIM In$[160]
20 INPUT In$
30 Length=LEN(In$)
40 DISP Length;"characters in """;In$;""""
50 END
```

Try finding the length of a string containing only spaces. When the `INPUT` statement is used, any leading or trailing spaces are removed from items typed on the keyboard. Change `INPUT` to `LINPUT` in line 20 to allow leading and trailing spaces to be entered.

Maximum String Length

The MAXLEN function returns an integer equal to the dimensioned length of a string variable. For example:

```
100 DIM First_string$[37],Second_string$(2)[15]
110 PRINT "Maximum length of the first string is";
120 PRINT MAXLEN(First_string$)
130 PRINT
140 PRINT "Maximum length of the second string is";
150 PRINT MAXLEN(Second_string$(1))
160 Test("A TEST STRING")
170 END
180 SUB Test(A$)
190 PRINT
200 PRINT "Maximum length of the test string is";
210 PRINT MAXLEN(A$)
220 SUBEND
```

5 Result:

```
Maximum length of the first string is 37
```

```
Maximum length of the second string is 15
```

```
Maximum length of the test string is 13
```

Substring Position

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

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

Result: 4

If POS returns a non-zero value, the entire substring occurs in the first string and the value specifies the starting position of the substring.

Note that POS returns the first occurrence of a substring within a string. By adding a subscript, and indexing through the string, the POS function can be used to find all occurrences of a substring. The following program uses this technique to extract each word from a sentence.

5-10 String Manipulation

```

10 DIM A$(80)
20 A$="I know you think you understand what I said, but you don't."
30 INTEGER Scan,Found
40 Scan=1 ! Current substring position
50 PRINT A$
60 REPEAT
70   Found=POS(A$(Scan)," ") ! Find the next ASCII space
80   IF Found THEN
90     PRINT A$(Scan,Scan+Found-1) ! Print the word
100    Scan=Scan+Found ! Adjust "Scan" past last match
110   ELSE
120     PRINT A$(Scan) ! Print last word in string
130   END IF
140 UNTIL NOT Found
150 END

```

As each occurrence is found, the new subscript specifies the remaining portion of the string to be searched.

String-to-Numeric Conversion

5

The VAL function converts a string expression into a numeric value. The string must evaluate to a valid number or an error will result. The number returned by the VAL function will be converted to and from scientific notation when necessary. For example:

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

Result: 123400

The following program converts a fraction into its equivalent decimal value.

```

10 INPUT "Enter a fraction (i.e. 3/4)",Fraction$
20 !
30 ON ERROR GOTO Err
40 Numerator=VAL(Fraction$)
50 !
60 IF POS(Fraction$,"/") THEN
70   Delimiter=POS(Fraction$,"/")
80   Denominator=VAL(Fraction$[Delimiter+1])
90 ELSE
100  PRINT "Invalid fraction"
110  GOTO Quit
120 END IF
130 !

```

```

140     PRINT Fraction$;" = ";Numerator/Denominator
150     GOTO Quit
160 Err: PRINT "ERROR Invalid fraction"
170     OFF ERROR
180 Quit: END

```

Similar techniques can be used for converting: feet and inches to decimal feet or hours and minutes to decimal hours.

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

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

Result: 65

The next program prints the value of each character in a name.

```

10     INPUT "Enter your first name",Name$
20     !
30     PRINT Name$
40     PRINT
50     FOR I=1 TO LEN(Name$)
60         PRINT NUM(Name$[I]); ! Print value of each character
70     NEXT I
80     PRINT
90     END

```

5

Entering the name: JOHN will produce the following.

```
74 79 72 78
```

Numeric-to-String Conversion

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

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

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

The next program converts a number into a string so the **POS** function can be used to separate the mantissa from the exponent.

5-12 String Manipulation

```

10 CONTROL 2,0;1 ! CAPS LOCK ON
20 INPUT "Enter a number with an exponent",Number
30 !
40 Number$=VAL$(Number)
50 !
60 PRINT Number$
70 E=POS(Number$,"E")
80 IF E THEN
90 PRINT "Mantissa is",Number$[1;E-1]
100 PRINT "Exponent is",Number$[E+1]
110 ELSE
120 PRINT "No exponent"
130 END IF
140 END

```

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$(12)           Clear screen
PRINT CHR$(7)           Ring the bell
PRINT CHR$(97);CHR$(98);CHR$(99)

```

5

Prints: abc

CRT Character Set

The following program prints the character set on the screen of the CRT to show the order that strings will be sorted.

```

10 ! Program: CRT Character Set.
20 !
30 PRINT CHR$(12);"CRT Character Set"
40 STATUS 1,9;Line_length ! 50, 80, or 128 Columns
50 Left=Line_length/2-16
60 !
70 FOR I=0 TO 255
80 Col=I MOD 16*2+Left
90 Row=I DIV 16+3
100 IF Col=Left THEN
110 PRINT TABXY(Left-5,Row);
120 PRINT USING "3D";I
130 END IF
140 PRINT TABXY(Col,Row);
150 CONTROL 1,4;1 ! Display Functions on

```

```

160     PRINT USING "B,B,#";128,I ! Print the Character
170     CONTROL 1,4;0          ! Display Functions off
180     NEXT I
190     PRINT
200     I=127
210     ON KNOB .08 GOSUB Change
220     DISP USING "5A,5D,X,2A,B,B";"ASCII",I,"=",128,I
230     GOTO 220
240 Change:   I=I-KNOBX/10
250           IF I<0 THEN I=0
260           IF I>255 THEN I=255
270           RETURN
280 END

```

ASCII character values from 128 to 159 are treated differently by different systems. Refer to the section “The Extended Character Set” found in this chapter.

5

String Functions

This section covers string functions which perform the following tasks:

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

String Reverse

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

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

Prints: `snac kcanS`

A common use for the `REV$` function is to find the last occurrence of an item in a string.

5-14 String Manipulation

```

10  DIM List$[30]
20  List$="3.22 4.33 1.10 8.55 12.20 1.77"
30  Length=LEN(List$)
40  Last_space=POS(REV$(List$)," ") ! "SPACE" is delimiter
50  DISP "The last item is: ";List$[1+Length-Last_space]
60  END

```

Displays: The last item is: 1.77

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.

```

10  INPUT "Enter your full name",Name$
20  First$=TRIM$(Name$[1,POS(Name$," ")])
30  Last$=TRIM$(Name$[1+LEN(Name$)-POS(REV$(Name$)," ")])
40  PRINT Name$,LEN(Name$)
50  PRINT Last$,LEN(Last$)
60  PRINT First$,LEN(First$)
70  END

```

Note that the `INPUT` statement trims leading and trailing blanks from whatever is typed. If you need to enter leading or trailing spaces, use the `LINPUT` statement.

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.

```
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
```

5

Copying String Arrays and Subarrays

`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 “(*)” is not 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 an array name.

A subarray can be copied into another subarray of the same size and shape. For example, suppose you want to copy the string elements in a two-dimensional string array found in rows 1 through 3 and columns 5 and 6

5-16 String Manipulation

of the string array called `Sub_array$` into the array called `New$`, you would execute the following statement:

```
MAT New$= Sub_array$(1:3,5:6)
```

where the above statement assumes an `OPTION BASE` of 1 and that `New$` is dimensioned to be a 3×2 string array.

For more information on copying numeric and string arrays see the `MAT` statement in the *HP BASIC Language Reference*.

Searching and Sorting

Information stored in a string array often requires sorting. There are over a dozen common algorithms that may be used. Each algorithm has certain advantages depending on the number of items to be sorted, the current order of the items, the time allowed to sort the items, and the complexity of the algorithm.

A list of items can be sorted very quickly by the `MAT SORT` statement.

```
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
```

Sorting by Substrings

A substring range can be appended to the end of a MAT SORT *key specifier*. For example, to sort the entire first column of a two-dimensional string array called `Str_ary$` using the 3rd and 4th characters of each string, you would use this *key specifier*: `(* ,1)[3,4]`. The MAT SORT statement would be as follows:

5

```
MAT SORT Str_ary$(*,1)[3,4]
```

Items will then be sorted by the characters within the substring specified. No error results from specifying a substring position beyond the current length of the string.

```
10  PRINT CHR$(12)    ! Program: SUBSORT
20  DATA 1 OLD ORANGE,2 TINY TOADS,3 TALL TREES,4 FAT FOWLS,5 FRIED FISH
30  DATA 6 SLOW SNAILS,7 SLIMY SLUGS,8 AWFUL HOURS,9 NASTY KNIVES
40  DIM Things$(1:9)[38]
50  READ Things$(*)
60  First=1
70  Length=1
80  DISP "Use KNOB and SHIFT-KNOB to change sort field."
90  ON KNOB .2 GOTO Slide
100 Go:MAT SORT Things$(*)[First;Length]
110  FOR I=1 TO 9
120    PRINT TABXY(10,I);Things$(I);RPT$(" ",3)
130  NEXT I
140 W:GOTO W
150  !
160 Slide:STATUS 2,10;Shift    ! Check for SHIFT OR CTRL
170  S=SGN(KNOBX)
180  IF Shift THEN
190    Length=Length+S*(S>0 AND Length<16)+S*(S<0 AND Length>1)
200  ELSE
210    First=First+S*(S>0 AND First<18)+S*(S<0 AND First>1)
```

5-18 String Manipulation

```

220 END IF
230 DISP "MAT SORT Things$(*)[";First;";";Length;"]"
240 PRINT TABXY(9,10);RPT$(" ",First);RPT$("^^",Length);RPT$(" ",10)
250 GOTO Go
260 END

```

Adding Items to a Sorted List

Lists of strings can be maintained in sorted order. Every time a new item is added to the list, the list is sorted by the **MAT SORT** statement. To prevent overwriting any of the items already in the list, items should be added to the top (first array element) of a list sorted in ascending order and to the bottom (last array element) of a list sorted in descending order.

```

10 PRINT CHR$(12)
20 ! Since arrays are in COM, they "remember" old values.
30 ! After running, execute SCRATCH C to clear the arrays.
40 !
50 COM Ascend$(1:18)[18],Descend$(1:18)[18]
60 Again:I=I+1
70 INPUT "Enter a word",Word$
80 Ascend$(1)=Word$ ! Fill array at top
90 Descend$(18)=Word$ ! Fill array at bottom
100 CALL See
110 IF I<18 THEN Again
120 BEEP
130 END
140 !-----
150 SUB See ! DISPLAY THE ARRAYS
160 COM Ascend$(*),Descend$(*)
170 MAT SORT Ascend$ ! <- ascending sort
180 MAT SORT Descend$ DES ! <- descending sort
190 FOR J=1 TO 18
200 PRINT TABXY(1,J);RPT$(" ",49)
210 PRINT TABXY(1,J);J;TABXY(11,J);Ascend$(J);TABXY(31,J);Descend$(J)
220 NEXT J
230 SUBEND

```

5

Sorting by Multiple Keys

When sorting a multi-dimensional numeric or string array, it is possible to specify more than one key. The array will be sorted by the first key then the second key and so on until the key specifiers are exhausted. Once the first key sorts items into similar groups, the items within a group can be arranged in any order you choose.

```
10  COM Tool$(1:8,1:3)[10]
20  DATA PENCIL,RED,35,PENCIL,BLUE,12,PENCIL,GREEN,0,PENCIL,BLACK,17
30  DATA PEN,BLACK,17,PEN,BLUE,127,PEN,RED,55,PEN,GREEN,43
40  READ Tool$(*)
50  PRINT
60  PRINT "*** UNSORTED LIST ***"
70  Display
80  PRINT "*** SORT BY COLOR ***"
90  MAT SORT Tool$(*,2)[1,3]      ! Sort color by first three letters.
100 Display
110 PRINT "*** SORT BY COLOR THEN BY NAME ***"
120 MAT SORT Tool$(*,2),(*,1)    ! Two key sort.
130 Display
140 PRINT "*** SORT BY NAME THEN BY COLOR ***"
150 MAT SORT Tool$(*,1),(*,2)[1;3] DES
160 Display
170 END
180 !-----
190 SUB Display
200   COM Tool$(*)
210   K=K+1
220   FOR I=1 TO 8
230     FOR J=1 TO 3
240       PRINT Tool$(I,J),
250     NEXT J
260   PRINT
270 NEXT I
280 SUBEND
```

5

Sorting to a Vector

It is possible to determine the sorting order of items in an array without disturbing the array. This is accomplished by “sorting” to a single-dimensioned numeric array (vector). The vector will then contain the subscripts of the items in the order that the items would have been arranged.

```
10  DIM Month$(1:12)[3],Fix(1:12)
20  DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
30  READ Month$(*)
40  MAT SORT Month$ TO Fix    ! Sort to vector
50  PRINT Month$(*)
60  PRINT Fix(*)
70  FOR I=1 TO 12
80    PRINT Month$(Fix(I)),  ! Print months alphabetically
90  NEXT I
100 END
```

Running this program produces:

```
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
4 8 12 2 1 7 6 3 5 11 10 9
APR AUG DEC FEB JAN JUL JUN MAR MAY NOV OCT SEP
```

The first element of the vector contains a four (4), indicating the fourth element in the array would be the first element if the array were actually sorted.

Reordering an Array

The rows and columns of multiple dimension arrays can be reordered. Reordering is made according to a reorder vector (single dimension array). The vector contains the values of the subscripts of the array. When the array is reordered, the columns (or rows) are arranged according to the the order of the subscripts in the reorder vector. See the following program for an example of reordering.

Searching for Strings

The following program outlines a method for replacing a word in a string.

```
100 ! Program: Word_Replace
110 !
120 DIM Text$(80)
130 !
140 Search$="bad"
150 Replace$="good"
160 Text$="I am a bad string."
170 !
180 PRINT Text$
190 S_length=LEN(Search$)
200 Position=POS(Text$,Search$)
210 IF NOT Position THEN Quit
220 !
230 Text$=Text$[1,Position-1]&Replace$&Text$[Position+S_length]
240 !
250 PRINT Text$
260 Quit: END
```

5

```
Print:      I am a bad string.
           I am a good string.
```

Searching String Arrays

Searching string arrays is similar to searching numeric arrays. For example, assume array `List$` contains a list of names and dollar amounts. The program shown next puts the data into the source array (`List$`). It then searches for a particular name and outputs the corresponding dollar amount.

```
100 OPTION BASE 1 ! Select option base.
110 DIM List$(4)[20] ! Dimension source array.
120 DATA BLACK BILL $100.00,BROWN JEFF $150.00
130 DATA GREEN JIM $200.00,WHITE WILL $125.00
140 READ List$(*) ! Read data into List$.
150 PRINT USING "20A,/" ;List$(*) ! Output the original list.
160 MAT SEARCH List$(*)[1,5],LOC("BROWN");Person ! Search proper
170 ! portion of each string in List$ for a
180 ! particular person.
190 PRINT
200 IF Person<=4 THEN
210 PRINT List$(Person)[1,5];": ";List$(Person)[13,20] ! Output
```

5-22 String Manipulation

```
220      !                               specified name and dollar amount.
230      END IF
240      END
```

In this program a `MAT SEARCH` is used to find the string which contains the required name. Once that string is found, the portion of it containing the dollar amount is displayed. Note that the substring specifier is used in the search and display statements. If you run this program, the following results are obtained.

```
BLACK BILL $100.00
BROWN JEFF $150.00
GREEN JIM $200.00
WHITE WILL $125.00
```

```
BROWN: $150
```

5

Number-Base Conversion

The two functions `IVAL` and `DVAL` convert a binary, octal, decimal, or hexadecimal string value into a decimal number. The `IVAL$` and `DVAL$` functions convert a decimal number into a binary, octal, decimal, or hexadecimal string value. The `IVAL` and `IVAL$` functions are restricted to the range of `INTEGER` variables (-32 768 thru 32 767). The `DVAL` and `DVAL$` functions allow “double length” integers and thus allow larger numbers to be converted (-2 147 483 648 thru 2 147 483 647).

If you are familiar with binary notation, you will probably recognize the fact that `IVAL` and `IVAL$` operate on 16-bit values while `DVAL` and `DVAL$` operate on 32-bit values.

The program starts by prompting for a decimal number to be entered. As the digits are typed, the number is displayed in each of the possible number bases. The softkey `(k5)` or `(f5)` lets you select the different number bases. Pressing the spacebar will clear the display.

Introduction to Lexical Order

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

Predefined lexical orders include: `ASCII`, `FRENCH`, `GERMAN`, `SPANISH`, `SWEDISH`, and `STANDARD`. You can create lexical orders for special applications. The `STANDARD` lexical order is determined by an internal keyboard jumper, set at the factory to correspond to the keyboard supplied with the computer. The setting can be determined by examining the proper keyboard status register (`STATUS 2,8;Language`). Thus, the `STANDARD` lexical order on a computer equipped with a French keyboard will actually invoke the `FRENCH` lexical order.

5

Why Lexical Order?

A common task for computers is to arrange (sort) a group of items in alphabetical order. However, “alphabetical order” for a computer is normally based on the character sequence of the American Standard Code for Information Interchange (`ASCII`) character set. While the `ASCII` character sequence is adequate for many English Language applications, most foreign language alphabets include accented characters which are not part of the standard `ASCII` character set but must be included in the sequence to correctly sort the characters used in the language.

How It Works

The `LEXICAL ORDER IS` statement modifies the collating sequence by assigning a new value to each character. The new value, called a sequence number, is used in place of the character’s `ASCII` value whenever characters are compared.

The ASCII Character Set

The ASCII set consists of 128 distinct characters including uppercase and lowercase alpha, numeric, punctuation, and control characters.

The table to the right shows the complete ASCII character set, as displayed on the CRT. Each character is preceded by its ASCII value. The character's value is actually the decimal representation of the binary value (bit pattern) used internally, by the computer, to represent the character.

The characters are arranged in ascending value, which is to say, in ascending lexical order. A character is "less than" another character if its ASCII value is smaller. From the table it can be seen that "A" is less than "B" since the value of the letter "A" (65) is less than the value of the letter "B" (66).

If you have experimented with string comparisons based on the ASCII collating sequence, you may have noticed a few shortcomings. Consider the following words.

RESTORE, RE-STORE, and RE_STORE

Sorting these items according to the ASCII collating sequence will arrange them in the following order.

RE-STORE < RESTORE < RE_STORE

This points out a limitation of string comparisons based on ASCII sequence. Since the hyphen's value (45) is less than any alpha-numeric character, and the underbar's value (95) is greater than all uppercase alpha characters, a word containing a hyphen will be less than the same word without the hyphen, and a word containing an underbar will be greater than the same word without the underbar. The **LEXICAL ORDER IS** statement lets you overcome these limitations by changing the sorting order of the character set.

Displaying Control Characters

Several special display features are available through the use of **STATUS** and **CONTROL** registers. Normally, ASCII characters 0 through 31 (control characters) are not displayed on the CRT. To enable the display of control characters, execute the following statement.

```
CONTROL 1,4;1 OR DISPLAY FUNCTIONS ON
```

Printing a line of text to the CRT will now show the trailing carriage-return and linefeed. Although this mode is useful for some applications, control characters are usually not displayed on the CRT.

CONTROL 1,4;0 or DISPLAY FUNCTIONS OFF

Turns off the special display functions mode.

The Extended Character Set

Only 128 characters are defined in the ASCII character set. An additional 128 characters are available in the extended character set. The extended set includes CRT highlighting characters, special symbols, and Roman Extension characters (accented vowels and other characters used in many European languages).

Note



Some printers produce different extended characters than those displayed on the CRT. Check the printer manual for details on alternate character sets.

5

Display Enhancement Characters

Certain combinations of characters sent to the display using PRINT or DISP affect the way characters are displayed. The characters which control underlining, inverse video, and blinking are **display enhancement characters**. These characters do not actually appear on the display themselves; they affect the appearance of subsequent printable characters. Whether or not any of these display enhancements are available depends on the capabilities of your display hardware.

For a complete list of all BASIC display enhancement characters, refer to the tables in the back of the *HP BASIC Language Reference* and the *HP BASIC Condensed Reference*.

SYSTEM\$ can be used to determine what CRT highlights are present. The expression

```
SYSTEM$("CRT ID")
```

returns a string containing information such as the CRT width and available highlights. The string returned by this expression is for Series 300 medium resolution monochrome monitors is:

```
6: 80H GB1
```

The 80 is the width of the CRT in characters and the H indicates that monochrome highlights are available. If there were a space instead of the H, then the CRT does not have highlights.

You can also determine if you have CRT highlights by sending highlight characters to the CRT and seeing if anything happens. For example, CHR\$(255) and CHR\$(132) turns on underlining and CHR\$(255) and CHR\$(128) turns off enhancements. Thus, on a display with highlights, the following:

```
PRINT CHR$(132);"This is important.";CHR$(128)
```

produces this:

```
This is important. On a display with highlights, this text is underlined.
```

On a display without highlights, the display enhancement characters are ignored and the line is displayed as normal text. Note that these display enhancement characters produce an action only in PRINT and DISP statements. When viewed in EDIT mode or on the system message line, these display enhancement characters appear as "h_p" or as shown in the previous table "Extended Character Set for CRT."

Alternate CRT Characters

There is a keyboard control register for the CRT mapping of character codes, changing the contents of the register may cause different characters to be displayed.

Try the following.

```
PRINT CHR$(247)
CONTROL 1,11;1
PRINT CHR$(247)
CONTROL 1,11;0
```



The first print statement will produce the character expected from the character tables. The second print statement may show a character (double arrow) from an alternate character set. Note that the alternate character set is only available on some displays.

Finding "Missing" Characters

By now, you may have noticed that there are more possible CRT characters than keys on the keyboard. If your particular keyboard does not have a key for the character you need, locate the `ANY CHAR` key (every keyboard has this key).

When you press the `ANY CHAR` key, the message, "Enter 3 digits, 000 to 255" appears in the lower left corner of the CRT. Enter the three digits: 065 and the character whose value is 65 (the letter "A") will be placed on the screen. Any character can be input by this method. Pressing a non-digit key or entering a value outside the range will cancel the function.

Predefined Lexical Order

5 When the LEX Binary is first loaded or after a `SCRATCH A`, the computer executes a `LEXICAL ORDER IS STANDARD` statement. This will be the correct lexical order for the language on the keyboard. This can be checked by examining the keyboard status register (`STATUS 2,8;Language`) or by either of the following statements.

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

The table below shows the language indicated by the value returned by the `STATUS` statement. Thus, if the value returned indicates a French keyboard, the `STANDARD` lexical order will be the same as the `FRENCH` lexical order. The `STANDARD` lexical order for the Katakana keyboard is ASCII.

Value	Keyboard Language	Lexical Order
0	ASCII	ASCII
1	FRENCH	FRENCH
2	GERMAN	GERMAN
3	SWEDISH	SWEDISH
4	SPANISH (European Spanish keyboard)	SPANISH
5	KATAKANA	KATAKANA
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 (Latin Spanish keyboard)	SPANISH
15	DANISH	SWEDISH
16	FINNISH	SWEDISH
17	NORWEGIAN	SWEDISH
18	SWISS FRENCH*	FRENCH
19	SWISS GERMAN*	GERMAN
20	KANJI	¹

¹ Refer to the *HP BASIC 6.2 Porting and Globalization* manual for information about the KANJI character set.

Note


The predefined lexical-order tables are found in the “Useful Tables” section in the *HP BASIC Language Reference* manual. For information about user-defined lexical orders, refer to the *HP BASIC 6.2 Advanced Programming Techniques* manual.

Either the CHR\$ function or **ANY CHAR** may be used to produce characters not readily available on the keyboard.

Subprograms and User-Defined Functions

This chapter describes the benefits of using subprograms, and shows many of the details of using them.

An Example

The following program contains two subprograms and one user-defined function:

```

10  OPTION BASE 1
20  DIM Numbers(20)
30  CALL Build_array(Numbers(*),20)  ! Subprogram call.
40  CALL Sort_array(Numbers(*),20)  ! Subprogram call.
50  PRINT FNSum_array(Numbers(*),20) ! User function call.
60  END
65  !
70  SUB Build_array(X(*),N)          ! Subprogram "Build_array".
80    ! X(*) is the array to be defined
90    ! N tells how many elements are in the array
100   ! (1 is assumed to be the lower index)
110   FOR I=1 TO N
120     DISP "ELEMENT #";I;
130     INPUT "?",X(I)
140   NEXT I
150  SUBEND
155  !
160  SUB Sort_array(A(*),N)          ! Subprogram "Sort_array".
170    ! A(*) is array to be sorted
180    ! N tells how many elements are in the array (1 is assumed
190    !   to be the lower bound)
200    ! Sort the array (elements 1-N) in increasing order
210    ! Algorithm used: Shell sort or Diminishing increment sort
220    ! Ref: Knuth, Donald E., The Art of Computer Programming,
```

6


```

230 ! Vol. 3 (Sorting and Searching), (Addison-Wesley 1973)
240 ! pp. 84-85
250 INTEGER T,S,H,I,J
260 REAL Temp
270 T=INT(LOG(N)/LOG(2)) ! # of diminishing increments
280 FOR S=T TO 1 STEP -1
290   H=2^(S-1) ! ... 16,8,4,2,1
300   FOR J=H+1 TO N
310     I=J-H
320     Temp=A(J)
330   Decide: IF Temp>=A(I) THEN Insert
340   Switch: A(I+H)=A(I)
350     I=I-H
360     IF I>=1 THEN Decide
370   Insert: A(I+H)=Temp
380   NEXT J
390 NEXT S
400 SUBEND
405 !
410 DEF FNSum_array(A(*),N) ! User-defined function "Sum_array".
420 ! Add A(1) ... A(N)
430 INTEGER I
440 REAL Array_total
450 FOR I=1 TO N
460   Array_total=Array_total+A(I)
470 NEXT I
480 RETURN Array_total
490 FNEND

```

6

Lines 10 through 60 are the main program. As you can see, it does nothing but call subprograms, which in turn do all the work. Line 70 is the header for the subprogram which asks the user to enter the values stored in his array. Notice that the main program has declared the array's name to be Numbers(*), but the subprogram uses the name X(*) to deal with the same array. The subprogram can name its variables whatever it wants without interfering with variables used outside the subprogram's context. The only variables that can be affected outside the subprogram's context are those passed through the parameter list (as shown here) or through COM (discussed later). In both cases, the matching between the subprogram and the outside world is done through the position of the variable(s) in the parameter list or COM block, not the actual name of the variable(s).

6-2 Subprograms and User-Defined Functions

Starting at line 160 is the next subprogram which sorts the array into ascending order. The comments at the front of the subprogram serve to discuss the definition of the parameters used, and what effect the subprogram has on them. Also, the algorithm used is given, along with the proper reference material. It is an excellent idea to give a list of such pertinent details at the front of all subprograms. This makes debugging, modifying, optimizing, and re-using the subprogram much easier.

Starting at line 410 we see an example of a function subprogram. Functions are similar to SUB subprograms in concept. This particular example just adds the elements of the array together and returns the final value to the main program, which prints it.

A Closer Look at Subprograms

The preceding examples only showed some of the general features of subprograms. This section shows a few of the details of using subprograms.

Calling and Executing a Subprogram

We have seen in the above examples how the two types of subprograms are called—SUBs 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. Thus our example of the main program which causes an array of numbers to be sorted could look like this:

```
10  OPTION BASE 1
20  DIM Numbers(20)
30  Build_array(Numbers(*),20)
40  Sort_array(Numbers(*),20)
50  PRINT FNSum_array(Numbers(*),20)
60  END
```

There are, however, three instances which require the use of CALL when invoking a subprogram.

CALL is required:

1. If the subprogram is called from the keyboard,
2. If the subprogram is called after the THEN keyword in an IF statement, or
3. In an ON..event..CALL statement.
4. When using procedure variables (CALL A\$)

Differences Between Subprograms and Subroutines

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

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

6

If you are a newcomer to HP BASIC, be careful to distinguish between these two terms. They have been used differently in some other programming languages.

Subprogram Location

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

Subprogram and User-Defined Function Names

A subprogram has a name which may be up to 15 characters long. Here are some legal subprogram names:

```
Initialize  
Read_dvm  
Sort_2_d_array  
Plot_data
```

Difference Between a User-Defined Function and a Subprogram

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

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

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

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)  
PRINT FNascii_to_hex$(A$)
```

In general, if you want to analyze data and generate a single value, then you probably want to implement the subprogram as a function. 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, it is better to use a SUB subprogram.

Program/Subprogram Communication

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

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

Parameter Lists

There are two places where parameter lists occur:

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

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

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

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

6

```
70 SUB Build_array(X(*),N) ! Subprogram "Build_array".
410 DEF FNSum_array(A(*),N) ! User-defined function "Sum_array".
```

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

Formal Parameter Lists

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

- The *number of values* that may be passed to a subprogram
- The *types of those values* (string, INTEGER, 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.)

6-6 Subprograms and User-Defined Functions

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

Pass Parameter Lists

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

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

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

Passing By Value vs. Passing By Reference

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

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

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

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

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

The general rules for passing parameters are as follows:

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

- In order for a parameter to be passed *by value*, the pass parameter list must use an *expression* for that parameter.

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

Example Pass and Corresponding Formal Parameter Lists

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

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

6

@Dvm	This is an I/O path name which may refer to either an I/O device or a mass storage file. Its name here implies that it is a voltmeter, but it is perfectly legal to redirect I/O to a file just by using a different ASSIGN with @Dvm.
A(*)	This is a REAL array. Its size is declared by the calling context. 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 Upper	These are declared here to be INTEGERS. Thus, when the calling program invokes this subprogram, it must supply either INTEGER variables or INTEGER expressions, or an error will occur.
Status\$	This is a simple string that returns the status of the voltmeter to the main program. The length of the string is defined by the calling context.
Errflag	This is a REAL number. The declaration of the string Status\$ has limited the scope of the INTEGER keyword which caused Lower and Upper to require INTEGER pass parameters.

6-8 Subprograms and User-Defined Functions

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

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

@Voltmeter	This is the pass parameter which matches the formal parameter @Dvm in the subprogram. I/O path names are always passed by reference, which means the subprogram can close the I/O path or assign it to a different file or device.
Readings(*)	This matches the array A(*) in the subprogram's formal parameter list. Arrays are always passed by reference.
1, 400	These are the values passed to the formal parameters Lower and Upper . Since constants are classified as expressions rather than variables, these parameters have been passed by value. Thus, if the subprogram used either Lower or Upper on the left-hand side of an assignment operator, no change would take place in the calling context's value area.
Status\$	This is passed by reference. If it were enclosed in parentheses, it would be passed by value. Notice that if it were passed by value, it would be totally useless as a method for returning the status of the voltmeter to the calling context.
Errflag	This is passed by reference.

6

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. 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.)

The OPTIONAL/NPAR combination is very effectively used in situations requiring external instrument setups. Most instruments have several different ranges, modes, settings, etc., which can be used depending upon the requirements of the user. Often, the user doesn't require the entire flexibility the instrument has to offer, and would rather use some reasonable defaults.

Consider the HP 3437A Digital Voltmeter. Among other things, this device has two data formats (packed and ASCII), three trigger modes (internal, external, and hold/manual), three voltage ranges (0.1V, 1V, and 10V), and also has programmable values for delay between readings and number of readings taken. Naturally, the values used for the various settings will depend entirely upon the application for which the voltmeter is being used, but let's make some assumptions:

- The values for delay and number of readings are going to be changed frequently, so they will not be OPTIONAL parameters.
- Of the remaining OPTIONAL parameters, the range is most likely to be altered.

A reasonable setup routine for the voltmeter might look like this:

```
2010 SUB Setup_dvm(@Dvm,INTEGER Readings,REAL Delay, OPTIONAL INTEGER
Prange, Ptrigger,Pformat)
2020 SELECT NPAR
2030 CASE 3
2040     Format=1                ! Default ASCII format
2050     Trigger=1             ! Default internal trigger
2060     Range=2               ! Default 1 volt range
2070 CASE 4
2080     Format=1
2090     Trigger=1
2100     Range=Prange
2110 CASE 5
2120     Format=1
2130     Trigger=Ptrigger
2140     Range=Prange
2150 CASE 6
2160     Format=Pformat
2170     Trigger=Ptrigger
2180     Range=Prange
2190 END SELECT
2200 OUTPUT @Dvm;"N"&VAL$(Readings)&"SD"&VAL$(Delay)
&"SR"&VAL$(Range)&"T"&VAL$(Trigger)&"F"&VAL$(Format)
```

6

6-10 Subprograms and User-Defined Functions

```
2210 SUBEND
```

Legal invocations of the Setup_dvm subprogram are:

```
570 Setup_dvm(@Dvm,100,.001)      ! Default Range,Trigger,Format
630 Setup_dvm(@Dvm,500,.05,3)     ! Default Trigger,Format
850 Setup_dvm(@Dvm,50,.005,1,2)   ! Default Format
1010 Setup_dvm(@Dvm,70,.075,2,1,2) ! Explicitly declare all values
```

Notice in the example above that local variables are used instead of the formal parameters. This is because it is illegal to use an OPTIONAL parameter variable if that variable was not passed from the calling context.

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),Subvalves(10,15),@Valve_ctrl
```

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

Any context need only declare those COM blocks which it needs to 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.

COM vs. Pass Parameters

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

- COM survives pre-run. In general, any numeric variable is set to 0, strings are set to the null string, and I/O path names are set to undefined after pushing the **(RUN)** key, or upon entering a subprogram. This is true of COM the first time the **(RUN)** key is pressed, but after COM block variables are defined, they retain their values until:
 - SCRATCH A or SCRATCH C is executed,
 - A statement declaring a COM block is modified by the user, or
 - A new program is brought into memory using the GET 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. This can impose a restriction on the size of your parameter lists.

6

COM blocks can take as many statements as necessary. COM statements can be interwoven with other statements (though this is considered a poor practice). All COM statements within a context which have the same name will be part of the definition of that COM block.

- COM blocks can be used for communicating between contexts that do not invoke each other. Information such as modes and states can be an integral part of communicating between contexts, even though those contexts don't explicitly call each other. For instance, one routine might be responsible for setting the voltage range on a voltmeter, while another routine may need to know what the current voltage range is in order to set up the scale on a graph properly.

6-12 Subprograms and User-Defined Functions

- COM blocks can be used to communicate between subprograms that are not in memory simultaneously. Similar to the case above, subprograms can communicate with each other through COM blocks even though combinations of LOADSUB/DELSUB may preclude their simultaneous presence in memory.
- COM blocks can be used to retain the value of “local” variables between subprogram calls. In general, the variables used by a subprogram are discarded when the subprogram is exited. However, there are situations where it might be useful for a subprogram to “remember” a value. A machine which tests capacitors in an incoming inspection department may require calibration after every 100 tests are performed. If the subprogram which does the testing has a way to count how many tests it has already performed (using a labeled COM block), then this task can be left to the testing routine, simplifying the rest of the system.
- COM blocks allow subprograms to share data without the intervention of the main program. Subprogram libraries may consist of elaborate relationships of both programs and data structures. In many cases, a major portion of the data structures are only used for support of the task being performed, rather than being integral to the task itself. Thus the main program does not need to declare the supportive data structures.

Examples of this situation might include data base management libraries (hashing tables may need to be maintained for accessing data quickly) or three-dimensional graphics libraries (window, viewport, and clip information need to be kept, as well as object definitions and related transformations).

6

Hints for Using COM Blocks

Any COM blocks needed by your program must be resident in memory at prerun time (prerun 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.

6

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

Consider the following COM block definition:

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

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

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

6-14 Subprograms and User-Defined Functions

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$
```

The following declaration is illegal, since it uses explicit size specifications for the array and string which do not match the original definition from line 10.

```
5020 COM /Dvm_state/ INTEGER Range,Format,N,REAL Delay,Lastdata(1:30),Status$[15]
```

The following declaration is also illegal, since it violates the types set forth by the defining block.

```
6010 COM /Dvm_state/ Range,Format,N,REAL Delay,Lastdata(*),Status$
```

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

Calling Subprograms Using a String Name

In all the previous discussion, subprogram calls were made explicitly by specifying the subprogram name. For example, here are the ways you might call **Mysub** explicitly:

```
CALL Mysub(Distance)
Mysub(Distance)
```

This method is adequate in the vast majority of cases. As an advanced feature, BASIC also allows you to call a subprogram using a string name. For example, the following code segment is equivalent to previous call to **My_sub**:

```
100 Name$="Mysub"
110 CALL Name$ WITH (Distance)
```

For details about **CALL**, refer the *HP BASIC Language Reference*.

Context Switching

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

- The calling context has a `DATA` pointer which points to the next item in the current `DATA` block which will be used the next time a `READ` is executed (assuming of course that a `DATA` block even exists in the calling program). This pointer is saved away whenever a subprogram is called, and then the `DATA` pointer is reset to the first `DATA` statement in the new subprogram context.
- The `RETURN` stack for any `GOSUBs` in the current context is saved and set to the empty stack in the new context.
- The system priority of the current context is saved, and the called subprogram inherits this value. Any change to the system priority which takes place within the subprogram (or any of the subprograms which it calls in turn) is purely local, since the system priority is restored to its original value upon subprogram exit. This is an important consideration: if the subprogram is called as a result of an event-initiated `GOSUB/CALL` statement, any `ON <event> GOTO/GOSUB/CALL/RECOVER` condition set up in the called subprogram must have a higher priority assigned to it than the event responsible for the subprogram’s invocation. Otherwise, the event is guaranteed *not* to cause an end of line branch. See “Interrupts and Timeouts” (chapter 18) for a description of system priority.
- Any event-initiated `GOTO/GOSUB` statements are disabled for the duration of the subprogram. If any of the specified events occur, this will be logged, but no action will be taken. (The fact that an event did occur will be logged, but only once—multiple occurrences of the same event will not be serviced.) Upon exiting the subprogram, these event-initiated conditions will be restored to active status, and if any of these events occurred while the subprogram was being executed, the proper branches will be taken.

- Any event-initiated CALL/RECOVER statements are saved away upon entering a subprogram, but the subprogram still inherits these ON conditions since CALL/RECOVER are global in scope. However, it is legal for the subprogram to redefine these conditions, in which case the original definitions are restored upon subprogram exit.
- The current 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. 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



6

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.

It is important to remember that the called subprogram inherits the softkey labels. All the keys are still active in some sense; ON KEY ... CALL/RECOVER will cause their original program branches to take place immediately if the proper key is pressed, and ON KEY ... GOTO/GOSUB will log the fact that a key is pressed until the subprogram is exited, at

which time the proper branch will occur. This latter case may cause some consternation on the part of the user if he presses a softkey expecting immediate action and nothing happens since the key was temporarily disabled due to a called subprogram. If the called subprogram is expected to take a noticeably long time to execute, it might be a good idea to explicitly remove the labels from the disabled softkeys using the OFF KEY statement. Thus, the user won't expect anything to happen as a result of pressing a softkey. This technique is also useful for guaranteeing that a given subprogram is *not* interrupted prematurely. (The DISABLE statement is useful for preventing program branches as a result of an event-initiated happening, although it will not turn off the softkey labels.)

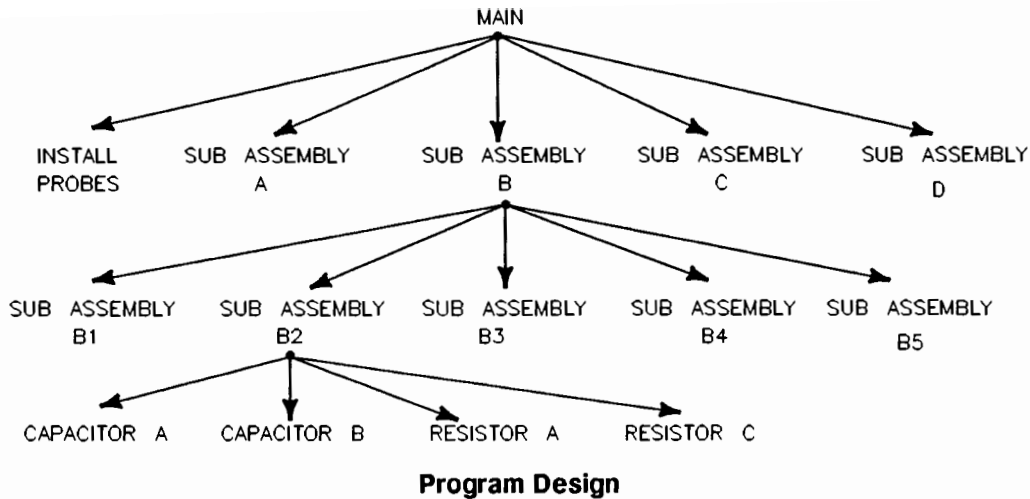
Subprograms and the RECOVER Statement

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.

By way of illustration, consider the following example. Suppose you are performing an exhaustive component test on a circuit board. The program may be designed like so:

6



When lunch break comes around, you may want to halt the current test so you can use the computer to play chess, or your boss might wander by and want to see the results of the rest of the tests performed this week. In either case, if the test program is nested three or four levels deep in subprograms, it might take a while for the test to complete. By defining a softkey to RECOVER to the main program, you can instantly terminate the test at any time, and make the computer available for something else. The RECOVER will discard anything being done in any of the subprograms between the context declaring the event-initiated RECOVER, and the subprogram being executed when the specified event occurs.

Again, the DISABLE statement can be used within any subprograms in which it is critical not to allow interruptions.

Calling Subprograms from the Keyboard

Functions and subprograms can be called by using CALL and FN 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

This section shows some of the tasks involved in using and managing subprogram libraries.

Why Use Subprogram Libraries?

6

As mentioned earlier, subprograms are also convenient for use in creating and distributing libraries. They are also handy when you have a large program, along with sizable data arrays, which could potentially require more memory than is currently installed in your computer. You can break the program up into subprograms, each of which may be programmatically loaded, called, and then deleted in order to conserve memory. This section outlines some of the operations you will need to perform in creating, using, and maintaining subprogram libraries.

Listing the Subprograms in a PROG File

You can determine which subprograms are in a PROG file by performing a CAT on the file:

```
CAT "ProgFile"
```

The system returns a list of the subprograms and user-defined functions in the file, along with other information, such as the amount of memory required for

6-20 Subprograms and User-Defined Functions

each subprogram. (See the *HP BASIC Language Reference* description of CAT for details.)

Loading Subprograms

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

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

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

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

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

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

```
LOADSUB FROM "File"
```

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

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

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

Loading Subprograms One at a Time

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

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

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

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

Loading Several Subprograms at Once

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

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

and if the user selects Option 2,

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

and so forth.

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

Loading Subprograms Prior to Execution

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

This can be handy in two ways. The first and obvious way is that subprograms can be loaded quickly. The other way is this: Type a LOADSUB FROM command where the file name is a file in which you *know* there are none of the

6-22 Subprograms and User-Defined Functions

subprograms you need (perhaps a null PROG file). Of course, no subprograms will be loaded, but *a list of those yet undefined will be printed.*

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

Note

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

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

6

Deleting Subprograms

The utility of the LOADSUB commands would be greatly reduced if you could not delete subprograms from memory. There is a way to delete subprograms during execution of a program: DELSUB. If you want to delete only selected ones, you could type something like this (you can also execute these statements in programs):

```
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:

```
DELSUB Print_report TO END
```

You can combine these methods:

```
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 non-existent subprogram, an error occurs, and the DELSUB statement is terminated. This means that subprograms whose names are listed after the error-causing name will not be deleted.

You can avoid getting this error by first determining if the statement is already in memory by executing the INMEM command. For example:

```
IF INMEM ("mysub")  
THEN DELSUB "mysub"
```

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 cannot delete itself.
2. A subprogram can not delete the subprogram that called it, either directly or indirectly.

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

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 you 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.

With the PDEV binary, the job is much easier:

1. Write your new subprogram *at the end* of the program.
2. Perform a MOVE LINES command where:
 - a. The Starting Line in the MOVE LINES command is the line which you want to immediately follow your new subprogram,
 - b. The Ending Line in the MOVE LINES command is the line immediately prior to the SUB or DEF FN of the new subprogram, and
 - 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 the **DEL LN** or **Delete line** key 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 MOVE LINES 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.

6

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 $N \times (N-1)!$ where $0!=1$ by definition. Thus $N!$ is simply the product of all the whole numbers from 1 through N inclusive. A recursive function which computes N factorial is:

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

We'll consider a more useful application of recursion in the following section on Top-Down Design.

6

Top-Down Design

A major problem that every programmer faces is designing programs that can be easily implemented and tested. A method of program design that has become widely recommended is Top-Down Design, also known as Stepwise Refinement.

The general approach is to consider a problem at its highest level, and break it down into a small number of identifiable subtasks. Each subtask is in turn considered as a large problem which is to be broken down into smaller problems, and so on until the “smaller problems” which have to be solved turn out to be lines of code, which the computer knows how to solve! At the higher levels of this process, the various subtasks are implemented as subprogram calls. It is best to define exactly what each subprogram is supposed to do long before the subprogram is actually written. Furthermore, this should be done at each level of refinement. By considering what each subprogram requires as input and what it returns as output from the topmost levels, the most serious problems of programming (namely defining your data structures and the communications paths between subprograms) are attacked at the beginning of the problem solving process, rather than at the end when all the small pieces are trying to jumble together. It is best to tackle these questions at the beginning because then you have the most flexibility—no code has been written and it’s not necessary to try and save any investment in programming time.

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 a program, **mass storage files** are more appropriate. Files provide means of storing data on mass storage devices.

Storing Data in Programs

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

Storing Data in Variables

The easiest 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
```

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`.

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
.
.
.
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.

7

Using DATA and READ Statements

The DATA and READ statements provide another technique for storing and retrieving data from the computer's 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.

7-2 Data Storage and Retrieval

For example:

```
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, 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. For example:

```
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. A READ into a COMPLEX variable is satisfied with two REAL DATA values. 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. The exception is when a COMPLEX variable is read—two numeric data items are consumed. If there are more variables than data items, the system returns an error, and the data pointer is moved back to the position it occupied before the READ statement was executed.

Examples

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

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

The date is November 26 1981

7

Storage and Retrieval of Arrays

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

7-4 Data Storage and Retrieval

REAL and COMPLEX DATA values can be assigned to elements of a 3-by-3 numeric array.

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

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

```
10  OPTION BASE 1
20  COMPLEX Example2(3,3)
30  DATA 23,-2,-1,10,-6,-7,4,5,-8,10,1,1,34,2,9,17,-12,-14
40  READ Example2(*)
50  PRINT USING "3(3D,X,3D,3X),/";Example2(*)
60  END
```

```
23 -2  -1  10  -6  -7
 4  5  -8  10   1   1
34  2   9  17 -12 -14
```

In the first example, the data pointer is left at item 10; thus, items 10 and 11 are saved for the next READ statement. In the second example, there are just enough items to fill each element of the complex array.

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.

```
100  DIM Array1(1:3)  ! Dimensions a 3-element array.
110  DIM Array2(0:4)  ! Dimensions a 5-element array.
120  DATA 1,2,3,4    ! Places 4 items in stream.
130  DATA 5,6,7      ! Places 3 items in stream.
140  READ A,B,C       ! Reads first 3 items in stream.
150  READ Array2(*)   ! Reads next 5 items in stream.
```



```

160 DATA 8,9          ! Places 2 items in stream.
170                   !
180 RESTORE           ! Re-positions pointer to 1st item.
190 READ Array1(*)    ! Reads first 3 items in stream.
200 RESTORE 140       ! Moves data pointer to item "8".
210 READ D            ! Reads "8".
220                   !
230 PRINT "Array1 contains: ";Array1(*);" "
240 PRINT "Array2 contains: ";Array2(*);" "
250 PRINT "A,B,C,D equal: ";A;B;C;D
260 END

```

```

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

```

File Input and Output (I/O)

This section describes the second general class of data storage and retrieval—mass storage files. It presents an overview of the available file types, and the BASIC programming techniques for accessing files. For a more detailed discussion of file I/O topics — including the internal data representation in files, the effect of record length, and end-of-file pointers — refer to “A Closer Look at File I/O” in the *HP BASIC 6.2 Advanced Programming Techniques* manual.

7 For background information about files and mass storage organization, refer to one of the following manuals, depending on the implementation of BASIC you are using:

- For BASIC/WS, refer to *Installing and Maintaining HP BASIC/WS 6.2*.
- For BASIC/UX, refer to *Installing and Maintaining HP BASIC/UX 6.2*.
- For BASIC/DOS, refer to *Installing and Using HP BASIC/DOS 6.2*.

7-6 Data Storage and Retrieval

Brief Comparison of Available File Types

HP BASIC provides four different types of files in which you can store and retrieve data.

Note



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

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

The main *disadvantages* of ASCII files are that:

- They can be accessed *serially* but not *randomly*.
- They can be written in *only default ASCII format* (no formatting is possible, and the data cannot be stored in internal representation).

Note



You can, however, format the data to be sent to an ASCII file by first sending it to a string variable with OUTPUT ... USING. You can then OUTPUT the string's formatted contents to a data file. Refer to the *HP BASIC 6.2 Advanced Programming Techniques* manual for details.

- BDAT—provide the most compact and flexible data storage mechanism. These files have several *advantages*:

- They can be *randomly or serially* accessed.
- More *flexibility* in data formats and access methods.
- *Faster* transfer rates.
- Generally more *space-efficient* than ASCII files (except for string data items).
- They allow data to be stored in ASCII format, internal format, or in a “custom” format (which you can define with IMAGE specifiers).

The *disadvantages* are that:

- You *must* know how the data items were written (as INTEGERS, REALs, COMPLEX values, strings, etc.) in order to correctly read the data back.
 - These data files cannot be *interchanged* with as many other systems as can ASCII files (for instance, the Series 200/300 Pascal Workstation system cannot read BDAT files).
- HP-UX—similar to BDAT files in structure, but also have some of the advantages of ASCII files:
 - Like BDAT files, they can also be accessed randomly or serially, and they can use ASCII, internal, or custom data representations.
 - Like ASCII files, they are useful for data-file interchange; however, the set of computers with which they can be interchanged is slightly different than LIF ASCII files. HP-UX files can be interchanged with any other system that uses the Hierarchical File System (HFS) format for mass storage volumes (such as HP-UX systems, and HP Series 200/300 Pascal systems beginning with version 3.2).
 - HP-UX files containing BASIC program lines can be read with GET and written with RE-SAVE.
 - DOS—similar to HP-UX files, but implemented by the DFS binary (for the HP Measurement Coprocessor) to provide file compatibility with MS-DOS. The DOS file type can exist only in a DFS directory.

In brief, choose a BDAT file for speed and compact data storage. Choose an ASCII, HP-UX, or DOS file for compatibility with operating systems and other computers.

Overview of File I/O

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

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

7

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

In order to store data in mass storage, a data file must be created (or already exist) on the mass storage media. In this case, line 440 creates a BDAT file (later sections describe using HP-UX and ASCII files); the file created contains 10 defined records of 256 bytes each. (Defined records and record size are discussed in the *HP BASIC 6.2 Advanced Programming Techniques* manual.)

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

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

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

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

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

7-10 Data Storage and Retrieval

The subsequent ENTER statements read the data items into variables; *with BDAT and HP-UX files* (when using the BASIC internal (FORMAT OFF) data representation), the *data type of each variable must match the data type type of each data item*. (This topic is discussed in the *HP BASIC 6.2 Advanced Programming Techniques* manual.) With ASCII files, for instance, you can read INTEGER items into REAL variables and not have problems.

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

A Closer Look at General File Access

Before you can access a data file, you must assign an I/O path name to the file. Assigning an I/O path name to the file sets up a table in computer memory that contains various information describing the file, such as its type, which mass storage device it is stored on, and its location on the media. The I/O path name is then used in I/O statements (OUTPUT, 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. Data transfers with devices and general I/O techniques are covered later in this manual. However, in this chapter we deal mostly with I/O paths to files.

Every I/O path to a file maintains the following information:

Validity Flag	Tells whether the path is currently opened (assigned) or closed (not assigned).
Type of Resource	Holds the file type: ASCII, BDAT, or HP-UX.
Device Selector	Stores the device selector of the drive. (I/O paths can also be associated with devices and buffers. See chapters 17 and 19 for further details.)
Attributes	Such as FORMAT OFF and FORMAT ON, BYTE, and PARITY ODD.
File Pointer	There is a file pointer that points to the place in the file where the next data item will be read or written. The file pointer is updated whenever the file is accessed.
End-Of-File Pointer	An I/O path has an EOF pointer that points to the byte that follows the last byte of the file.

Opening an I/O Path

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

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

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

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

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

7-12 Data Storage and Retrieval

opens an I/O path to the file **Example** on an HP 9122 disk drive, interface select code 7 and primary address 0. You must include the protect code or password, if the LIF or SRM file has one, respectively.

ASSIGNing an I/O path name to a file has the following effect on the I/O path table:

- If the I/O path is currently open, the system *closes* the I/O path and then *re-opens* it. If the I/O path is not currently open, it is opened. In both cases, the system sets the validity flag to Open.
- The file's type (ASCII, BDAT, or HP-UX) is set.
- The file's directory path (if in a hierarchical directory structure) and msus are recorded.
- The specified attributes are assigned to the I/O path name. If an attribute is not specified, the appropriate default attribute is assigned (such as FORMAT OFF with BDAT and HP-UX files, and FORMAT ON with ASCII files).
- The file pointer is positioned to the beginning of the file.
- If the I/O path name is associated with a BDAT or HP-UX file, the physical EOF pointer (read from the volume on which the file resides) is copied to the I/O path table.

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

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



Assigning Attributes

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

- With FORMAT ON, ASCII data representations are used.

- With `FORMAT OFF`, the BASIC system's internal data representations are used.

Additional attributes are available, which provide control of such functions as parity generation and checking, converting characters, and changing end-of-line (EOL) sequences. See `ASSIGN` in the *HP BASIC Language Reference*, or "I/O Path Attributes" (chapter 19) in this manual for further details.

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

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

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

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

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

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

7

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

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

Alternatively, you could re-enter the entire statement:

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

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

7-14 Data Storage and Retrieval

It is important to note that once a file is written, changing the **FORMAT** attribute of an I/O path to the file should only be attempted by experienced programmers. *In general, data should always be read in the same manner as it was written.* For instance, data written to a **BDAT** or **HP-UX** file with **FORMAT OFF** should also be read with **FORMAT OFF**, and vice versa. In addition, the same data types should be used to write the file as to read the file. For instance, if data items were written as **INTEGERS**, they should also be read as **INTEGERS** (this is mandatory with **FORMAT OFF**, but not always necessary with **FORMAT ON**).

In theory, there is no limit to the number of I/O paths you can **ASSIGN** to the same file. Each I/O path, however, has its own file pointer and **EOF** pointer, so that in practice it can become exceedingly difficult to keep track of where you are in a file if you use more than one I/O path. *We recommend that you use only one I/O path at any one time for each file.*

Closing I/O Paths

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

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

```
ASSIGN @Path2 TO *
```

closes **@Path2** (sets the validity flag to **Closed**). **@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.

7

Locking Files

Although sharing files between people saves disk space, it introduces the danger of several people trying to access a file at the same time. For instance, one person may read a file while another person is writing to it, and the file's contents may be inaccurate.

LOCK establishes exclusive access to a file: it can only be accessed from the workstation where the **LOCK** was executed. The typical procedure is:

1. LOCK all critical files.
2. Read data from files.
3. Update the data.
4. Write the data into the files.
5. UNLOCK all critical files to permit shared access again.

Example Locking and Unlocking of an SRM File

In this example, a critical operation must be performed on the file named `File_a`, and you do not want other people accessing the file during that operation.

```
1000  ASSIGN @File TO "File_a:REMOTE"
1010  LOCK @File;CONDITIONAL Result_code
1020  IF Result_code THEN GOTO 1010 ! Try again
1030  ! Begin critical process
      .
      .
      .
2000  ! End critical process
2010  UNLOCK @File
```

The numeric variable called `Result_code` is used to determine the result of the LOCK operation. If the LOCK operation is successful, the variable contains 0. If the LOCK is not successful, the variable contains the numeric error code generated by attempting to lock the file.

7

BASIC/UX Specifics on Locking SRM Files

BASIC/UX allows several instances of HP BASIC to be run on the same system. However, the SRM server assumes that each client is running on a different system. Because of this, the semantics of LOCK are different if more than one BASIC/UX process is trying to access the same file at the same time. An ENTER normally hangs until the file is unlocked. If the file is locked by another BASIC/UX process on the same system, however, the ENTER returns error 481.

Locking HFS Files (BASIC/UX ONLY)

BASIC/UX supports locking and unlocking of HFS files in addition to SRM files. In general, the behavior of LOCK and UNLOCK for HFS is similar to the behavior for SRM. HFS LOCK attempts to exclusively lock the whole file, if possible. It also keeps track of nested locks, and all locks are released when the file is closed. In addition, if the file is already locked, the LOCK command does not block.

However, there are some differences due to the locking mechanism in the HP-UX operating system.

- The HP-UX operating system supports both “exclusive” and “advisory” locks. Advisory locks only prevent other processes trying to lock a file that you have already locked. However, it does not prevent other processes from reading or writing the file.
- The HP-UX operating system supports both “read locks” and “write locks”. Placing a read lock allows other processes to lock the file for reading, but not for writing. Placing a write lock prevents other processes from placing read locks or write locks. Therefore, in HP-UX terms, a truly exclusive lock is an “enforcement mode write lock” on the entire file.

To exclusively lock a file, the following conditions must be met:

- The file must not have any locks on it already.
- The file must have the permission bits set to allow enforcement mode locking (**set group id** bit true, and **group search** (execute) bit false) or you must own the file (BASIC/UX sets the bits automatically).

Unless the first condition is met, the file will not be locked. If the second condition is not met, an advisory mode lock will be placed instead of an enforced mode lock.

In addition, the following limitations apply to HFS locking:

- LOCK cannot be used on named or unnamed pipes. Attempting to do so will cause error #810, “unsupported on hp-ux”.
- The HP-UX operating system does not support locking of files accessed across the network using RFA or NFS services. Therefore LOCK has no effect when applied to files accessed in this way. LOCK does, however, work with diskless systems.

- HFS locks lock the file to the process which placed the lock. This is somewhat different from SRM locks on the single user BASIC/WS, which lock the file to the workstation.
- TRANSFER to a locked file is supported, however, there is a brief moment in time when the transfer starts and stops during which the lock can be lost.
- BASIC/UX uses the `lockf(2)` HP-UX operating system call to place the lock. See `lockf(2)` and `fcntl(2)` for additional information.

Note

For more advanced file I/O programming techniques, refer to “A Closer Look at File I/O” (chapter 4) in the *HP BASIC 6.2 Advanced Programming Techniques* manual.

Extended Access of Directories

The CAT statement has the following additional capabilities:

- Catalog an individual PROG-type file
- Send the directory to a string array
- Select files to be cataloged by name or by beginning letter(s) of the file name
- Count the number of selected file entries
- Skip a specific number file entries before sending entries to the destination
- Suppress the catalog header
- Use the CRT format when when sending the directory to a string array
- A listing of only the names of the files in the current working directory of the current default volume
- Catalog all files whose names match a given wildcard expression

7

Cataloging to a String Array

The following example program segment shows an example of directing the catalog of mass storage file entries to the CRT and then to a string array.

```
100 PRINT "          CAT to CRT."
110 PRINT "-----"
120 CAT TO #CRT;COUNT Files_and_headr ! Includes 5-line header.
130 PRINT "Number of files=";Files_and_headr-5
140 PRINT
150 !
160 PRINT "          CAT to a string array."
170 PRINT "-----"
180 Array_size=Files_and_headr+2 ! Allow for 7-line header.
190 ALLOCATE Catalog$(1:Array_size)[80]
200 CAT TO Catalog$(*)
210 FOR Entry=1 TO Array_size
220   PRINT Catalog$(Entry)
230 NEXT Entry
240 PRINT "Number of files=";Array_size-7
250 PRINT
260 !
270 END
```

The program produces the following output.

```
          CAT to CRT.
-----
:INTERNAL
VOLUME LABEL: B9836
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
Data1 ASCII 3 256 16 12-Jan-87 12:30
Chap1 BDAT 3 256 20 13-Jan-87 8:00
Prog1 PROG 2 256 23 14-Jan-87 9:10
Chap2 BDAT 7 256 26 14-Jan-87 10:15
Prog2 PROG 2 256 33 14-Jan-87 12:30
Data2 ASCII 9 256 35 3-Mar-87 6:45
Number of files= 6
```

7

CAT to a string array.

```
-----  
:INTERNAL, 4  
LABEL: B9826  
FORMAT: LIF  
AVAILABLE SPACE: 892  
  
SYS FILE NUMBER RECORD MODIFIED PUB  
OPEN  
FILE NAME LEV TYPE TYPE RECORDS LENGTH DATE TIME ACC  
STAT  
=====
```

Data1	1		ASCII	3	256	12-Jan-87	12:30	MRW
Chap1	1	98X6	BDAT	3	256	13-Jan-87	8:00	MRW
Prog1	1	98X6	PROG	2	256	14-Jan-87	9:10	MRW
Chap2	1	98X6	BDAT	7	256	14-Jan-87	10:15	MRW
Prog2	1	98X6	PROG	2	256	14-Jan-87	12:30	MRW
Data2	1		ASCII	9	256	3-Mar-87	6:45	MRW

```
=====  
Number of files= 6
```

You may have noticed that the format for catalogs sent to string arrays (the second catalog listing) is different from catalogs sent to the PRINTER IS device. The format for catalogs sent to string arrays is the SRM catalog format, which requires that each array element must be dimensioned to hold at least 80 characters with this type of CAT operation. Again, the header contains 7 lines, not 5 as with catalogs sent to devices.

Getting an "Extended" Catalog of a LIF or HFS Disk

7

When you are cataloging an HFS or LIF directory to a string array, the catalog format is normally that of an SRM catalog. However, you can also specify an HFS- or LIF-format catalog listing with the following syntax:

```
100 CAT TO String_array$(*); EXTEND
```

For an explanation of the HFS catalog listing, see the *HP BASIC Language Reference* description of CAT.

Getting a Count of Files Cataloged

Including the keyword `COUNT` followed by a numeric variable returns the total number of file entries plus header lines to that variable; in the preceding example program, the variable `Files_and_headr` is used:

```
20  CAT TO #CRT;COUNT Files_and_headr ! Includes 5-line header.
```

In the above example, line 180 adds 2 to the variable `Files_and_headr` to compensate for the 7-line header which is sent instead of the usual 5-line header (the next section shows how to suppress the header) and stores the result in `Array_size`. `Array_size` is then used to direct the computer to `ALLOCATE` just enough space in a string-array variable to hold the directory listing. The program can then search the directory listing for further information, if desired.

If the `CAT` operation would not have filled the string array, the unused array elements would have been set to the null string (i.e., strings of length 0). If there are more catalog lines than string-array elements, the operation stops when the array is filled. No indication of the “overflow” is reported; the count returned is equal to the number of array elements.

Suppressing the Catalog Header

To suppress the catalog header, use the following syntax:

```
CAT;NO HEADER  
CAT TO String_array$(*);NO HEADER  
CAT "Prog_2";NO HEADER
```

Using `NO HEADER` suppresses the 5-line heading of a LIF catalog format or 7-line heading of an SRM or HFS catalog format. The catalog listing of a `PROG` file would be 4 lines shorter. The first line of each catalog listing contains the first directory entry, the second element contains the second entry, and so forth.

If the `COUNT` option is included, the count returned is the total number of selected files.

7

Cataloging Selected Files

The directory entries for files can be selectively obtained either by using the secondary keyword `SELECT` or by using wildcards. An example will be given using both methods. For both examples, assume that the directory contains the following entries:

```
:INTERNAL
VOLUME LABEL: B9836
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
Data1 ASCII 3 256 16 12-Jan-87 12:30
Chap1 BDAT 3 256 20 13-Jan-87 8:00
Prog1 PROG 2 256 23 14-Jan-87 9:10
Chap2 BDAT 7 256 26 14-Jan-87 10:15
Prog2 PROG 2 256 33 14-Jan-87 12:30
Data2 ASCII 9 256 35 3-Mar-87 6:45
Chap3 BDAT 6 256 45 3-Mar-87 7:15
Number of files= 7
```

CAT with the `SELECT` Secondary

Suppose that you want to catalog only files beginning with the letters "Prog". The following examples show how this may be accomplished. Notice that this is *not* the same operation as getting a catalog of a `PROG` file.

```
Beginning_chars$="Prog"
CAT;SELECT Beginning_chars$

CAT;SELECT "Prog",COUNT Files_and_headr
```

Result:

```
:INTERNAL, 4
VOLUME LABEL: B9836
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
Prog1 PROG 2 256 23 12-Jan-87 12:30
Prog2 PROG 2 256 33 13-Jan-87 8:00
```

The directory entries of the files beginning with the letters "Prog" are sent to the `PRINTER IS` device. In the second `CAT` statement above, the variable `Files_and_headr` is filled with the number of selected files found on the current default mass storage device plus 5 or 7 header lines. Both `CAT` statements above go to the `PRINTER IS` device (or file).

7-22 Data Storage and Retrieval

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"
```

Note that if any other files begin with the letters "Chap3", they will also be listed.

CAT with Wildcards

Suppose that you wanted to catalog only files which began with the letters "C" or "D" and did not end with the digit "3". The following examples show how this might be accomplished when HP-UX style wildcards are enabled. First, enable wildcard recognition using:

```
WILDCARDS UX;ESCAPE"\
```

Then catalog the directory using:

```
CAT "[CD]*[!3]"
```

```
CAT "[CD]*[!3]",COUNT Files_and_headr
```

Only 4 entries in the directory match this wildcard argument. The directory entries of the 4 entries are sent to the PRINTER IS device. In the second CAT above, the variable Files_and_headr is filled with the number of files on the current default mass storage device that match the wildcard argument plus 5 or 7 header lines. Both CAT statements above go to the PRINTER IS device (or file).

The following result would be sent to the system printing device.

```
:INTERNAL, 4
VOLUME LABEL: B9836
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
Data1 ASCII 3 256 16 12-Jan-87 12:30
Chap1 BDAT 3 256 20 13-Jan-87 8:00
Chap2 BDAT 7 256 26 14-Jan-87 10:15
Data2 ASCII 9 256 35 3-Mar-87 6:45
```

Using wildcards with CAT is much more powerful than using the SELECT secondary.

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 match a certain wildcard argument. The COUNT option directs the computer to return the number of selected files in the variable that follows the COUNT keyword.

```
CAT;COUNT Files_and_headr
CAT;SELECT "Data",COUNT Selected_files,NO HEADER
CAT"*[0-9]",COUNT Selected_files,NO HEADER
```

The first CAT operation returns a count of all files in the directory plus 5 or 7 header lines, since not including SELECT defaults to "select all files". The second operation returns a count of the specifically selected files. If HP-UX style wildcards are enabled, the third operation returns a count of all the files having names which end in a digit. Keep in mind that the number of selected files includes the number of files sent to the destination plus the number of files skipped, if any.

Catalogs sent to external devices in the LIF format have a five-line header; in SRM and HFS formats they have seven-line headers. Catalogs to string arrays are SRM format unless EXTEND is added. Catalogs of individual PROG files have a three-line header and a one-line trailer. If an "overflow" of a string array occurs, the count is set to the number of string-array elements plus the number of files skipped. If no entries are sent to the destination (because the directory is empty, or because no entries were selected, or because all selected entries were skipped), the value returned depends on whether there is a header. If there is no header, then zero (0) is returned. If there is a header, then the value returned is the size of the header plus the number following the skip option (the number requested to be skipped).

7

If an option is given more than once, only the last instance is used.

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
```

```
:INTERNAL, 4
VOLUME LABEL: B9836
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
BCD_ENTFMT ASCII 10 256 73 13-Jan-87 8:00
```

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

Including COUNT in the previous CAT operation (as shown below) returns a count of the selected files (plus header lines), not just the catalog lines sent to the destination. Remember that selected files includes all files skipped, if any. In this case, a value of 11 is returned, not 1 (or 6) as might be expected.

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

Note that if SKIP is included, the count remains the same (as long as at least one file is cataloged). If the number of files to be skipped equals the number of files selected, COUNT returns a value of zero.

```
CAT;SELECT "BCD",SKIP 6,COUNT Files_and_headr
```

The following program shows an example of looking at the files in a catalog by viewing a small “window” of files at one time. The technique is useful for decreasing the amount of memory required to hold a catalog listing in a string array.

```
100 ! Declare a small string array (7 elements).
110 DIM Array$(1:7)[80]
120 !
130 ! Send header to the array.
140 CAT TO Array$(*)
150 ! Print header.
160 FOR Element=1 TO 7
170 PRINT Array$(Element)[1,45]
180 NEXT Element
190 !
200 ! Now get 7-line "windows" and print files therein.
210 First_file=1 ! Begin with first file in directory.
220 REPEAT ! Send file entries to Array$ until last file sent.
230 !
240 ! Send files to Array$; SKIP files already printed;
```

7

```

250     ! return index (with COUNT) of last file sent to Array$.
260     CAT TO Array$(*) ;SKIP First_file-1,COUNT Last_file,NO HEADER
270     DISP "First file=" ;First_file ;" ; Last file=" ;Last_file
280     !
290     ! Print file entries (no entry printed when Last_file=0).
300     FOR Element=1 TO (Last_file-First_file)+1 ! (6 or less)+1.
310         PRINT Array$(Element)[1,45]
320     NEXT Element
330     !
340     First_file=Last_file+1 ! Point to next "window."
350     !
360     UNTIL Last_file=0 ! Until SKIP >= number of files.
370     !
380     END

```

HFS File Buffering (BASIC/UX only)

File buffering increases the efficiency and speed of accessing files. (Note that it is only available with files on HFS-formatted disks.)

What Is File Buffering?

File buffering means temporarily storing data to be sent to or received from a file in a memory buffer (instead of immediately writing to or reading from the disk). BASIC/UX puts data that is to be sent to a file in HP-UX's large cache buffer (in computer memory). Each file currently open has a portion of the buffer space in this cache. BASIC keeps putting data into the buffer cache until it is full, and then sends it to the disk all at once.

Note



The buffer is also "flushed" at other times; for instance, when a file is closed or when a program ends. The above explanation has been simplified for clarity. A complete list of situations when buffers are flushed is described in the subsequent section called "Flushing the File Buffer".

This buffering process saves time and disk wear by consolidating several small disk-write operations into a larger one. However, there is a risk that you may

lose data if power fails before data in the buffer has been written to the disk. You can also lose data if the system is shut down improperly.

Turning File Buffering On and Off

Use CONTROL register 9 to turn HFS file buffering on and off. For example:

```
ASSIGN @Io_path to "file"      must assign an I/O path
CONTROL @Io_path,9;1          turn buffering on
CONTROL @Io_path,9;0          turn buffering off
STATUS @Io_path,9;Status_var  check status of file buffering
```

If the value returned in Status_var is 1, then file buffering is on; if the value is 0, then file buffering is off.

Flushing the File Buffer

Flushing the buffer means updating the physical disk with all of the data in the file buffer. File buffers are flushed to the system's physical disk whenever you:

- Close a file (ASSIGN @Io_path TO *).
- Turn file buffering off (using CONTROL @Io_path,9;0).
- Terminate BASIC/UX normally (using QUIT).
- Execute the HP-UX sync command from within BASIC/UX (EXECUTE sync) or from within another process on the system.
- Start the HP-UX syncer from "/etc/rc", which periodically runs the sync command (default period between syncs is 30 seconds).

7

DFS File Buffering (BASIC/DOS only)

The DFS file system implements file buffering that is similar to HFS file buffering. Refer to "Additional I/O Path Registers for DFS" in chapter 7 of *Installing and Using HP BASIC/DOS 6.2* for a complete description of DFS file buffering.

Graphics Techniques

One of the most exciting features of your Series 200/300 computer is its graphics capability. The graphic techniques chapters introduce you to the powerful set of graphics statements in BASIC and teach you how to use them to produce pleasing output.

If you use BASIC/WS and you want to know what binary to load for a particular keyword or option, see the *HP BASIC Language Reference*. Note that you must load the BIN files named GRAPH and GRAPHX before you can enter most graphics statements. You may need to load other BIN files, depending on what computer you have. Refer to the *Installing and Maintaining HP BASIC* manual for information about BIN files. Finally, some of the example graphics programs require BIN files such as MAT that you might not readily associate with graphics.

If you have BASIC/WS, you should have a disk called *Manual Examples*. The part numbers for this disk vary depending on what disk drive you have, but the disk contents are identical. The *Manual Examples* disk contains many of the programs discussed in the graphics chapters, which you can load and run as you read through the chapters.

If you have BASIC/UX, the graphics examples are located in the `/usr/lib/rmb/demo/manex` directory.

The file name for an example program is cited when the program is discussed so that you know which program to run. These file names are identical for BASIC/WS and BASIC/UX.



Why Graphics?

Some data follow. As quickly as you can, determine if the overall trend is steady, rising or falling. Are there any periodic motions? If so, how many cycles are represented in the 50 points?

Set of Voltage Data

Time (sec.)	Voltage	Time (sec.)	Voltage
1	16.10	26	16.24
2	16.25	27	16.27
3	16.25	28	16.44
4	16.28	29	16.44
5	16.36	30	16.57
6	16.31	31	16.60
7	16.27	32	16.70
8	16.08	33	16.72
9	16.10	34	16.66
10	16.06	35	16.58
11	16.07	36	16.62
12	16.17	37	16.46
13	16.14	38	16.33
14	16.26	39	16.34
15	16.34	40	16.36
16	16.40	41	16.45
17	16.56	42	16.52
18	16.60	43	16.56
19	16.44	44	16.77
20	16.51	45	16.89
21	16.35	46	16.80
22	16.41	47	16.96
23	16.28	48	16.80
24	16.19	49	16.74
25	16.30	50	16.77

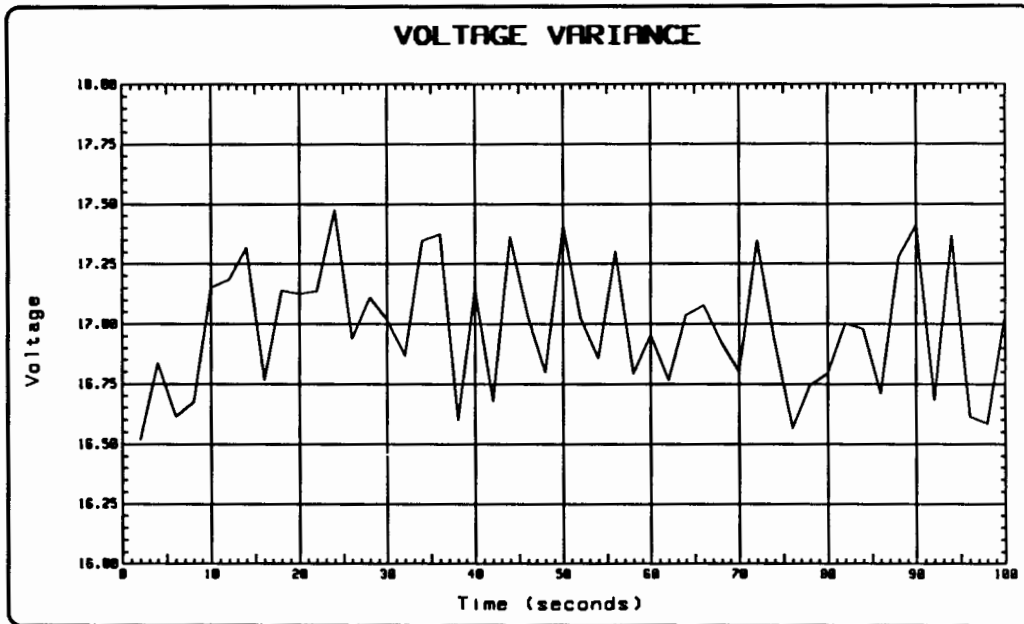
8

Following is a graph of the data in the preceding table. Observe that the graphical nature of the data makes the change in voltage over time clearer.

8-2 Graphics Techniques

Clarity and understandability at a glance is what computer graphics is all about.

Many example programs are included in the pages that follow. Type in and run them as you progress from simply drawing a jagged line to creating complex graphics.



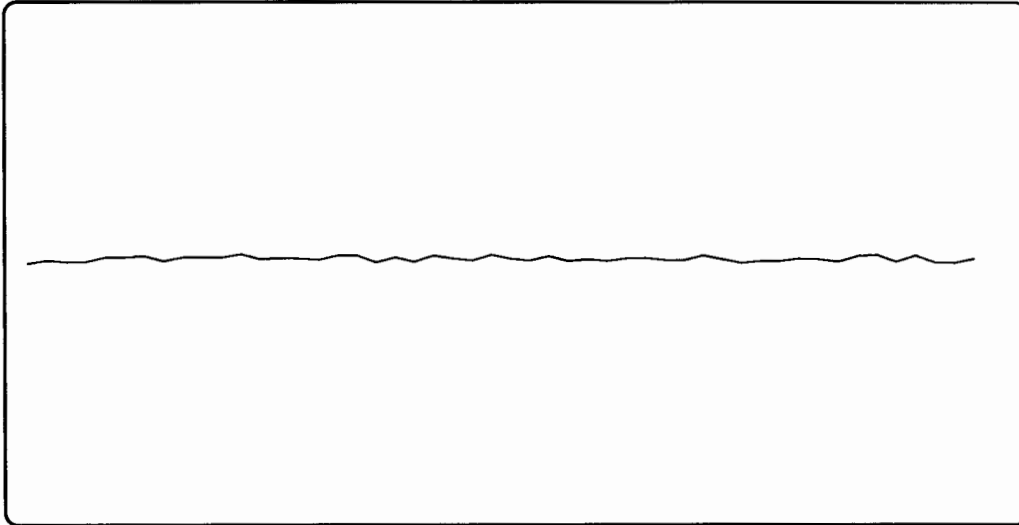
Graph of Voltage Data

Drawing Lines

To draw lines, you can simply PLOT the X and Y coordinates of the point you want to draw a line to. The following program does just that.

```
10  GINIT                                ! Initialize various graphics parameters.
20  PLOTTER IS CRT,"INTERNAL"            ! Use the CRT screen
30  GRAPHICS ON                          ! Turn on the graphics screen
40  FOR X=2 TO 100 STEP 2                ! Points to be plotted ...
50    PLOT X,RND+50                      ! Get a data point and plot it against X
60  NEXT X                               ! RND returns a value between 0 and 1
70  END                                  !
```

8



Example of Plotting Random Data

The GINIT statement on line 10 means *Graphics Initialize*. This is almost always the first graphics statement you execute. It sets various graphics parameters to their default values, and it is a shorthand way to execute up to 14 other statements (see the *HP BASIC Language Reference* manual for details).

The GRAPHICS ON statement on line 30 allows you to see what the program is drawing if you have separate alpha and graphics. On bit-mapped displays, graphics and alpha are always on, unless you have modified the display mask. More on this later.

Line 50 contains the heart of the program. In a loop, the PLOT statement draws to each successive point, which is determined by the loop control variable X for the X direction and the value returned by the function RND+50 for the Y direction. The constant, 50, centers the line on the screen so it is not displayed in your softkey display area.

8

8-4 Graphics Techniques

Scaling

The problem with the previous plot is that it doesn't show much information; it's too straight. If we exaggerated the Y direction to the point where we could see the variations, the line would better represent the plotted data.

This problem can be remedied by **scaling**. Scaling means defining the values the computer considers to be at the edges of the plotting surface. By definition, the left edge is the minimum X, the right edge is the maximum X, the bottom is the minimum Y, and the top is the maximum Y. Thus any point you plot which has X and Y coordinates within these ranges will be visible.

Two statements are available to define your own values for the edges of the plotting surface. The first one is **SHOW**, which forces X and Y units to be equal. 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 statement defines the plotting area such that there is a rectangle in that area with a minimum X=0, maximum X=100, minimum Y=16, and maximum Y=18, *using isotropic units*. As mentioned above, isotropic means that one unit in the X direction equals one unit in the Y direction. Thus, 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). This is because the X and Y units must be identical, so the **SHOW** statement centers the specified area in the plotting area, allowing whatever extra room it needs to ensure that the 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 defined the 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.

The next example uses a **SHOW** statement to define the edges of the screen to appropriate values. The X values used in the **SHOW** statement (0 and 100) represent 100 data points and indicate that axes are more meaningful when the origin is at zero and not 1. The Y values for this type of plot must be

determined by you or by the computer itself. We are using a random number function to simulate data received from some device.

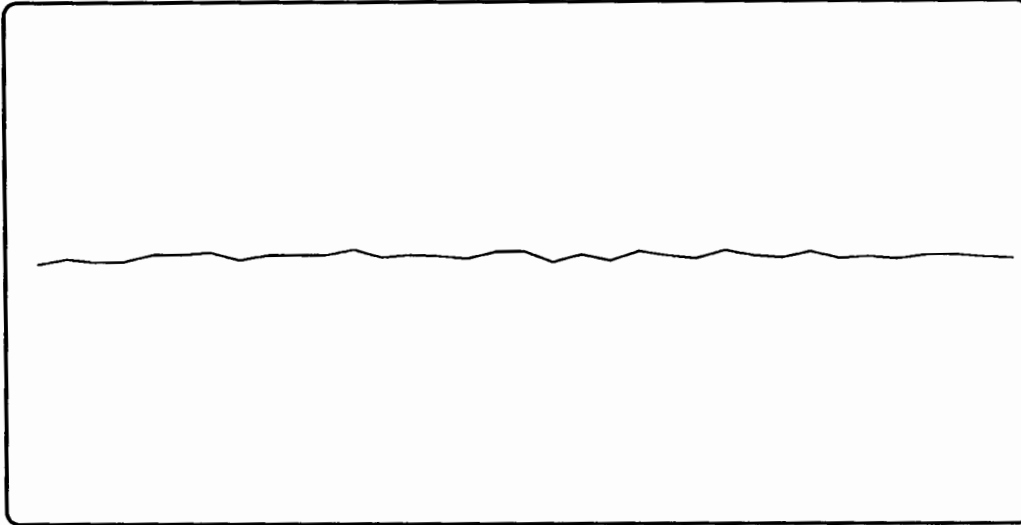
The MAX and MIN functions provide a method for determining the maximum and minimum values of a set of numbers. However, these functions require the MAT binary. For example:

```
110 Ymin=INT(MIN(Y(*)))
120 Ymax=MAX(Y(*))
130 Ymax=INT(Ymax)+(Ymax<>INT(Ymax))
```

Line 110 calculates the greatest integer less than or equal to the minimum value in an array of Y values. Lines 120 and 130 calculate the smallest integer greater than or equal to the maximum value in the array of Y values.

Back to our example, the Y values used (16 and 18) come from the RND function. In real applications, you probably will not know beforehand what the range of the data will be, in which case you can use the method described above to determine it.

```
10 GINIT ! Initialize various graphics parameters.
20 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
30 GRAPHICS ON ! Turn on the graphics screen
40 SHOW 0,100,15,19 ! Isotropic scaling: left, right, bottom, top
50 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
60 PLOT X,RND+17 ! Get a data point and plot it against X
70 NEXT X ! RND returns a value between 0 and 1
80 END !
```



Better Use of X-Axis Scaling

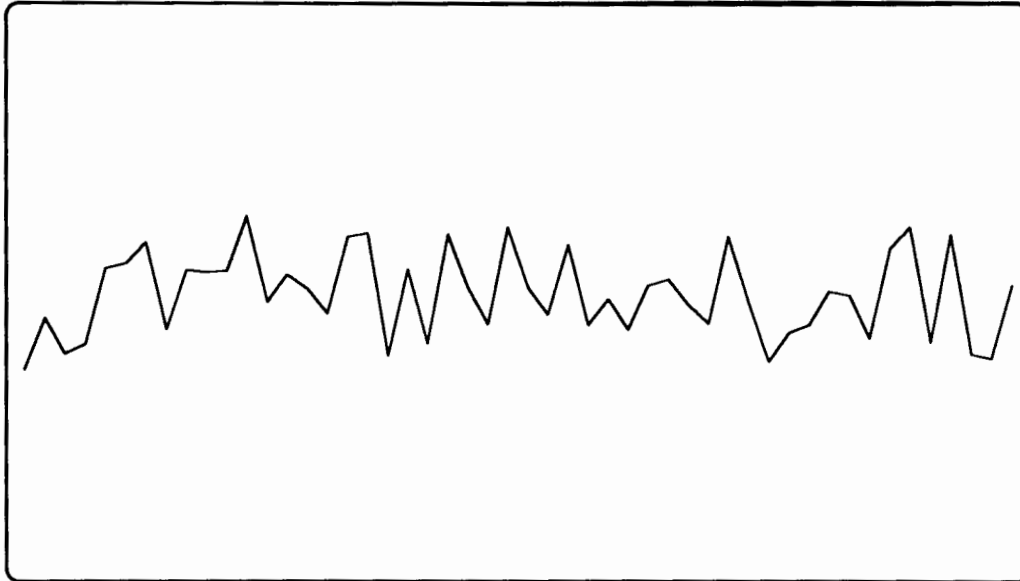
As you can see, SHOW centers the curve on the screen, but since the range of X values is so much larger than the range of Y values (0 to 100 versus 16 to 18), there is still not enough resolution to see what the data is doing.

In this example, we are dealing with data types (seconds and volts) that are not equal. Thus, it would be better to use unequal, or **anisotropic**, scaling to graph the relationship in the example. To do this, we can use the other scaling statement, WINDOW. WINDOW does not force X units to be equal to Y units. Instead, the axis range determines the scaling.

```

10  GINIT                ! Initialize various graphics parameters.
20  PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
30  GRAPHICS ON          ! Turn on the graphics screen
40  WINDOW 0,100,15,19   ! Anisotropic scaling: left,right,bottom,top
50  FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
60    PLOT X,RND+17      ! Get a data point and plot it against X
70  NEXT X               ! RND returns a value between 0 and 1
80  END                  !
  
```

8



Better Use of Y-Axis Scaling

In this plot, we can easily see variations in the data. To test how the Y axis range, 15-19, affects relative variations in the data, change `WINDOW 0,100,15,19` to `WINDOW 0,100,30,50` and change `RND+17` to `RND+40`. Run the program again and note that the line is less ragged (remember that `RND` ranges between 0 and 1).

There is still one problem, though. We can see *relative* variations in the data, but what units are being used—microvolts, millivolts, megavolts, dozens of volts, or what? We also 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 called the **soft clip area**. It is delimited by the **soft clip limits**. Clip, because line segments which attempt to go outside these limits are cut off at the edge of the subarea. Soft, because we can override these limits by turning off the clipping with the CLIP OFF statement (more about this later). There are **hard clip limits** also, which are defined to be the absolute limits of the plotting area. There is no way to override the hard clip limits.

GDUs and UDUs

Before we define a viewport, we need to know about the two different types of units that exist. These two types of units are UDUs (user-defined units) and GDUs (**graphics display units**). For viewports to be predictable, they must always be specified in the same units. Since users can change UDUs, GDUs are used 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 length of a GDU is defined as 1 percent of the shorter edge of the plotting area. The lower left of the plotting area is *always* 0,0. GDUs are isotropic; that is, one unit in the X direction is the same distance as one unit in the Y direction. For a CRT display, the Y-axis is always shorter, and thus it is 100 GDUs in length.

Unless you specify otherwise, the display screen (but *not* necessarily an external plotter) has the following expanse:

- In the Y direction (the shorter side): 0 through 100 GDUs.
- In the X direction: 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 this ratio is in the approximate range 1.25 to 1.33. To find this ratio for your display, just type RATIO Return. You can include the RATIO expression in another BASIC statement, for example:

```
VIEWPORT 0,RATIO*100,20,80
```


This practice ensures that the clipping limits you specify will work for any display type. Some approximate RATIO values for typical displays follow:

- 1.3344 (Models 216 and 226)
- 1.3136 (Models 236 and 236C)
- 1.2807 (HP 98542A and 98543A displays)
- 1.3338 (HP 98544, 98545, 98547, and 98700 displays)
- 1.2502 (HP 98550 and 98720 displays)
- 1.3340 (HP Measurement Coprocessor with VGA or EGA display)
- 1.3340 (Models 362 and 382 with 640-by-480 display)
- 1.3338 (Models 362 and 382 with 1024-by-768 display)

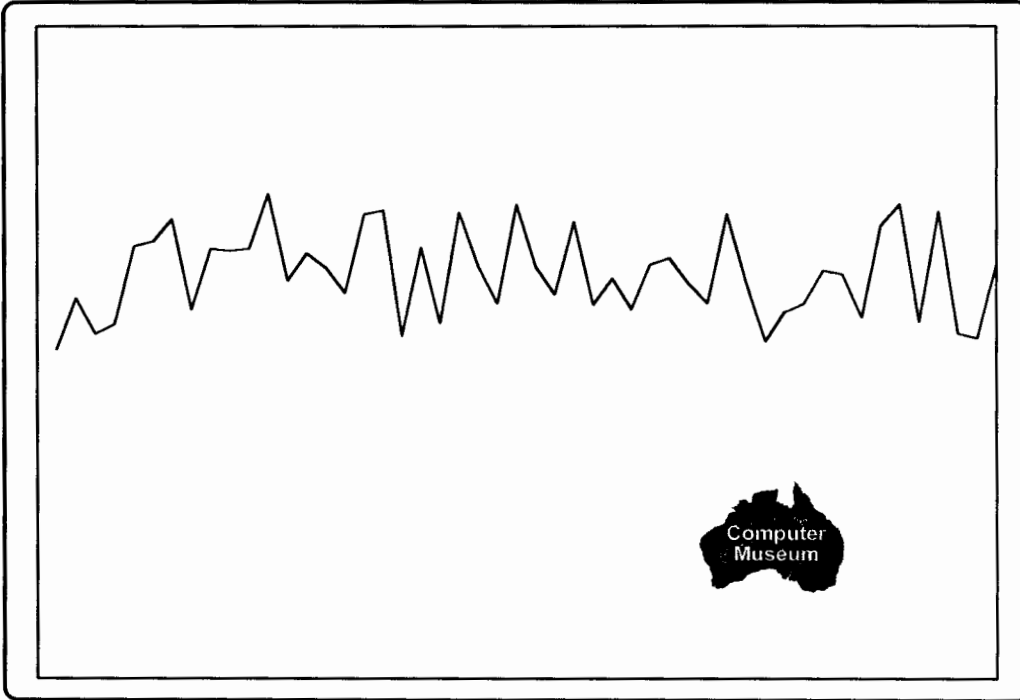
Thus, the length of the X axis is in the range of approximately 125 to 133 GDUs for any of the listed displays.

Specifying the Viewport

VIEWPORT 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 you can draw outside the subarea if you turn off clipping (more about clipping later). The VIEWPORT statement in the following 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 that the WINDOW statement affects. Also note line 50—the FRAME statement. This draws a box around the current soft clip limits so you can see the area specified by VIEWPORT.

```
10  GINIT                ! Initialize various graphics parameters.
20  PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
30  GRAPHICS ON          ! Turn on the graphics screen
40  VIEWPORT 10,120,15,90 ! Define subset of screen area
50  FRAME                ! Draw a box around defined subset
60  WINDOW 0,100,15,19   ! Anisotropic scaling: left,right,bottom,top
70  FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
80    PLOT X,RND+17      ! Get a data point and plot it against X
90  NEXT X               ! RND returns a value between 0 and 1
100 END                 !
```

8



Using VIEWPORT to Define Soft-Clip Limits

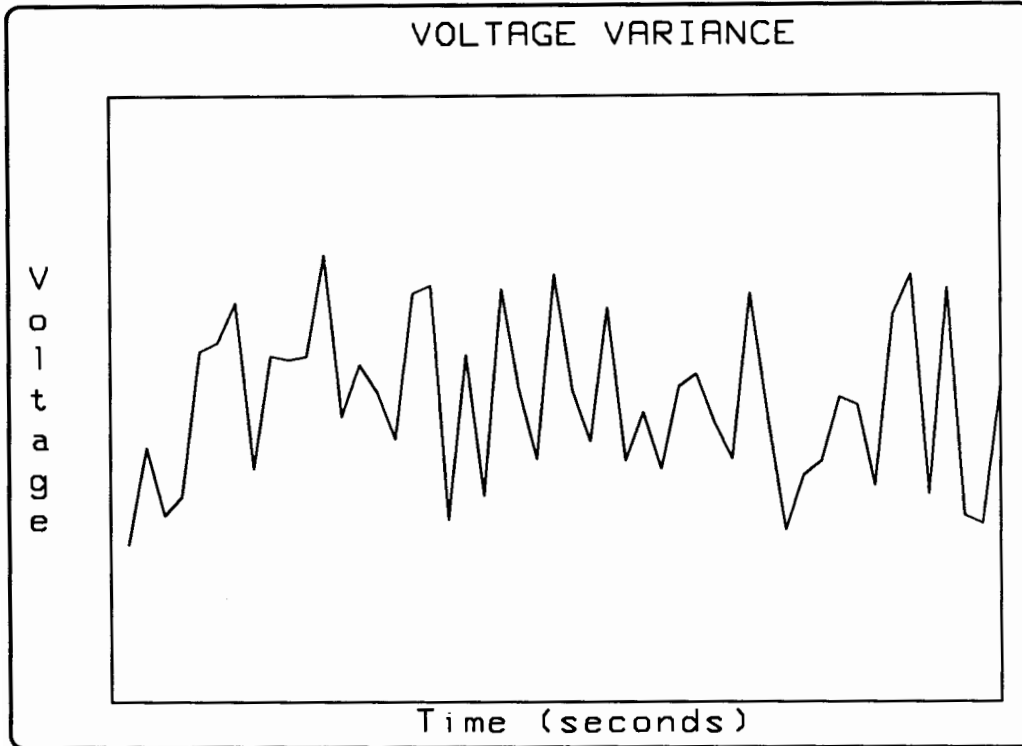
Labeling a Plot

By using VIEWPORT, we now have enough room to include labels on the plot. Typically, in a Y-vs.-X plot, a title for the whole plot is centered at the top, a Y-axis title is placed on the left edge, and an X-axis title is placed at the bottom.

LABEL lets you 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. The lower left corner of the label is at the point to which you moved. In other words, move to the position on the screen at which the lower left corner of the text is placed. Note that the LORG statement will move you to the lower left corner of the label. (The relative origin for labels can be changed with the LORG statement.)

Notice in the following plot that the Y-axis label on the left edge of the screen is created by writing one letter at a time. We 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. (There is a better way to plot vertical labels; we'll see it in the section titled "Data-Driven Plotting.")

```
10  GINIT                ! Initialize various graphics parameters.
20  PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
30  GRAPHICS ON          ! Turn on the graphics screen
40  MOVE 45,95           ! Move to left of middle of top of screen
50  LABEL "VOLTAGE VARIANCE" ! Write title of plot
60  MOVE 0,65           ! Move to center of left edge of screen
70  Label$="Voltage"     ! Write Y-axis label
80  FOR I=1 TO 7         ! Seven letters in "Voltage"
90    LABEL Label$[I,I]  ! Label one character
100 NEXT I              ! et cetera
110 MOVE 45,10          ! X: center of screen; Y: above key labels
120 LABEL "Time (seconds)" ! Write X-axis label
130 VIEWPORT 10,120,15,90 ! Define subset of screen area
140 FRAME                ! Draw a box around defined subset
150 WINDOW 0,100,16,18  ! Anisotropic scaling: left/right/bottom/top
160 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
170   PLOT X,RND*16.5    ! Get a data point and plot it against X
180 NEXT X              ! et cetera
190 END                 ! finis
```



Labeled Plot

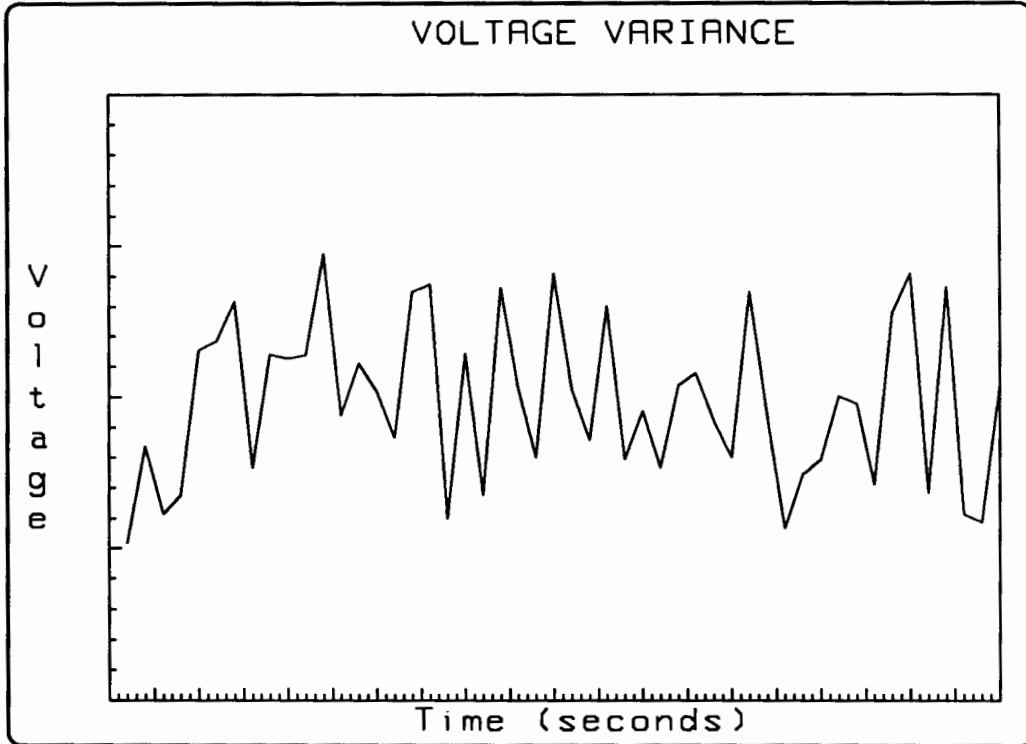
Now we know what we are measuring—voltage versus time—but we still do not know the units being used. We need an X-axis and a Y-axis labeled with numbers in appropriate places. The AXES statement will work here.

Axes and Tick Marks

The AXES statement draws X and Y axes and short lines perpendicular to the axes to indicate the unit spacing. These short lines are called **tick marks**. The axes may intersect at any desired point; it need not be the actual origin—0,0. 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 choose. Insert the **AXES** statement in your program and rerun it to see the difference.

```
10  GINIT                ! Initialize various graphics parameters.
20  PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
30  GRAPHICS ON          ! Turn on the graphics screen
40  MOVE 45,95           ! Move to left of middle of top of screen
50  LABEL "VOLTAGE VARIANCE" ! Write title of plot
60  MOVE 0,65            ! Move to center of left edge of screen
70  Label$="Voltage"     ! Write Y-axis label
80  FOR I=1 TO 7         ! Seven letters in "Voltage"
90    LABEL Label$[I,I]  ! Label one character
100 NEXT I               ! et cetera
110 MOVE 45,10           ! X: center of screen; Y: above key labels
120 LABEL "Time (seconds)" ! Write X-axis label
130 VIEWPORT 10,120,15,90 ! Define subset of screen area
140 FRAME                ! Draw a box around defined subset
150 WINDOW 0,100,16,18   ! Anisotropic scaling: left/right/bottom/top
160 AXES 1,.1,0,16,5,5,3 ! Draw X- and Y-axes with appropriate ticks
170 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
180   PLOT X,RND+16.5    ! Get a data point and plot it against X
190 NEXT X               ! et cetera
200 END                  ! finis
```



Plot with Axes and Tick Marks

Line 160 of the program contains the AXES statement and its parameters. The parameters are explained as follows:

- 1, .1 The X axis has 1 display unit between tick marks, and the Y axis has .1 display unit between tick marks, in current display units.
- 0, 16 The Y axis crosses the X axis at X=0. The X axis crosses the Y axis at Y=16.
- 5, 5 These counts are the number of "minor" (shorter) ticks between "major" (longer) tick marks on the axes. The value of 5 specifies that a major tick mark will appear every fifth tick mark.
- , 3 The value of 3 specifies that the major ticks are 3 GDUs long (the default is 2).

Using Graphics Effectively

The *Manual Examples* disk (`/usr/lib/rmb/demo/manex` for BASIC/UX users) contains programs found in this chapter. As you read through the following sections, load the appropriate program and run it. Remember that some of the programs require the MAT (matrix) BIN file. You can also type in the listed programs and run them. Either way, experiment with them until you are familiar with the demonstrated concepts and techniques.

More on Defining a Viewport

Recall that the VIEWPORT statement defines a subset of the screen in which to plot. More precisely, the VIEWPORT statement *defines a rectangular area into which the WINDOW coordinates will be mapped*. VIEWPORT immediately rescales the plotting area; thus, it is a good programming practice to follow every VIEWPORT statement with a WINDOW statement. The VIEWPORT also invokes clipping at its edges. There will be more about clipping later in this chapter.

The Y direction edge values default to 0 through 100 in Y. The X direction left edge value is 0. The right edge value can vary depending on what computer you have (approximately 128-133). Technically, these are UDUs, although default UDUs are equivalent to the GDUs until you change the UDUs with a SHOW or a WINDOW. To recap the important characteristics of GDUs:

- The lower left of the plotting area is 0,0.
- GDUs are isotropic.

As mentioned earlier in the section titled “Scaling,” it is trivial to determine how long the shorter edge of screen is in GDUs, but substantially more involved to calculate the length of the longer edge in GDUs. Since GDUs have X and Y units of the same length, the length in GDUs of the longer edges of the plotting surface is closely related to the **aspect ratio** of the plotting surface. The aspect ratio is the ratio of width to height of the plotting surface. A function called RATIO returns the quotient of these values.

Using RATIO, we can derive two formulas which are almost indispensable when writing a general VIEWPORT statement:

```
X_gdu_max=100*MAX(1,RATIO)
Y_gdu_max=100*MAX(1,1/RATIO)
```

These two statements define the maximum X and Y in GDUs. *This will work no matter what plotting device you are using.* Now that we have X_gdu_max and Y_gdu_max defined, we have complete control of the subset we want on the plotting surface. Suppose we want:

- the left edge of the viewport to be 10 percent of the hard clip limit width from the left edge,
- the right edge of the viewport to be 1 percent of the hard clip limit width from the right edge,
- the bottom edge of the viewport to be 15 percent of the hard clip limit height from the bottom, and
- the top edge of the viewport to be 10 percent of the hard clip limit height from the top.

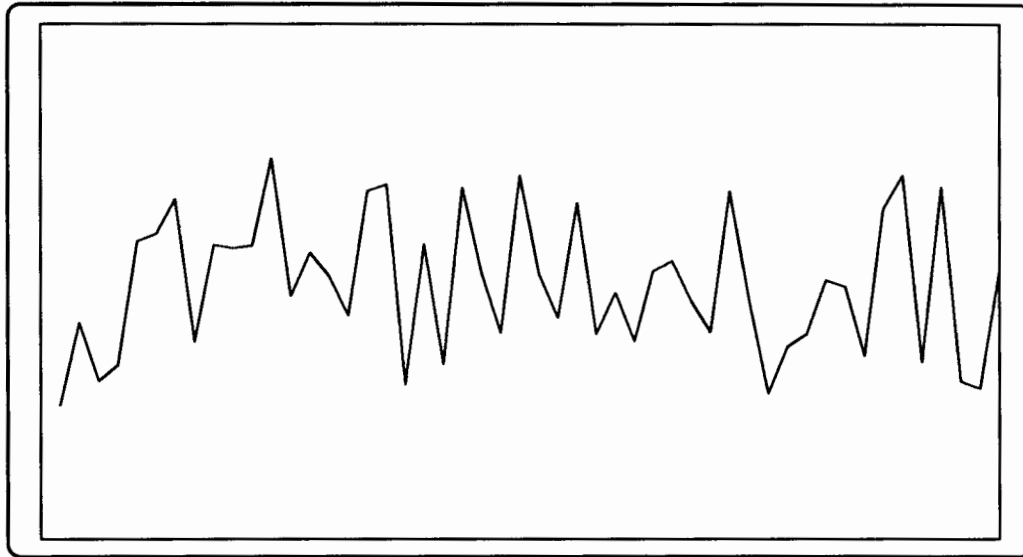
We would specify:

```
VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
```

Now, let's return to the program which defined the viewport, and update the VIEWPORT statement. (Run the example program in file SinViewprt.)

```
100 CLEAR SCREEN           ! Clear the alpha display
110 GINIT                  ! Initialize various graphics parameters.
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON           ! Turn on the graphics screen
140 X_gdu_max=100*MAX(1,RATIO) ! How many GDUs wide the screen is
150 Y_gdu_max=100*MAX(1,1/RATIO) ! How many GDUs high the screen is
160 VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
      ! Define subset of screen area
170 FRAME                  ! Draw a box around defined subset
180 WINDOW 0,100,16,18    ! Anisotropic scaling: left/right/bottom/top
190 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
200   PLOT X,RND+16.5     ! Get a data point and plot it against X
210 NEXT X                 ! RND returns a value between 0 and 1
220 END
```

8



General-Purpose VIEWPORT (SinViewprt)

Special Considerations about Scaling

If you scale the plotting area with WINDOW or SHOW using “large” coordinates, and then execute a MOVE, PLOT or DRAW command followed by another WINDOW or SHOW using “small” coordinates, a math overflow error may occur. What do we mean by “large” or “small” coordinates? Coordinates greater than 10^7 are considered large in this case, while coordinates less than 60,000 are considered small.

The following example program produces an overflow error:

```

10 PLOTTER IS CRT, "INTERNAL"
20 A = 5.4 * 10^7           ! Choose a "large" coordinate value
30 WINDOW 0,10,0,A        ! Scale the plotting area
40 MOVE 0,A               ! Move to edge of plotting area
50 WINDOW 0,10,0,10       ! Math overflow error occurs here
60 MOVE 5,5
70 END

```

To avoid this error, add line 45 as shown below:

```
10 PLOTTER IS CRT, "INTERNAL"  
20 A = 5.4 * 10-7           ! Choose a "large" coordinate value  
30 WINDOW 0,10,0,A         ! Scale the plotting area  
40 MOVE 0,A                ! Move to edge of plotting area  
45 MOVE 0,0                ! Avoids math overflow error  
50 WINDOW 0,10,0,10  
60 MOVE 5,5  
70 END
```

Why Does This Math Overflow Error Occur?

The graphics system does most of its math with real numbers until it is time to deal with actual pixel locations on the screen. (The word "pixel" is a blend of the two words "picture element." It is the smallest addressable point on a plotting surface.) At this time, it converts the real values to integers. When working with large values (such as 10^7), overflows can occur.

In the example above, line 40 moves the graphics pen to the edge of a very large plotting area which is defined in REAL coordinates. When the plotting area is rescaled to much smaller (REAL) values in line 50, the location of the graphics pen does not change. In other words, the graphics pen retains its large REAL coordinate values.

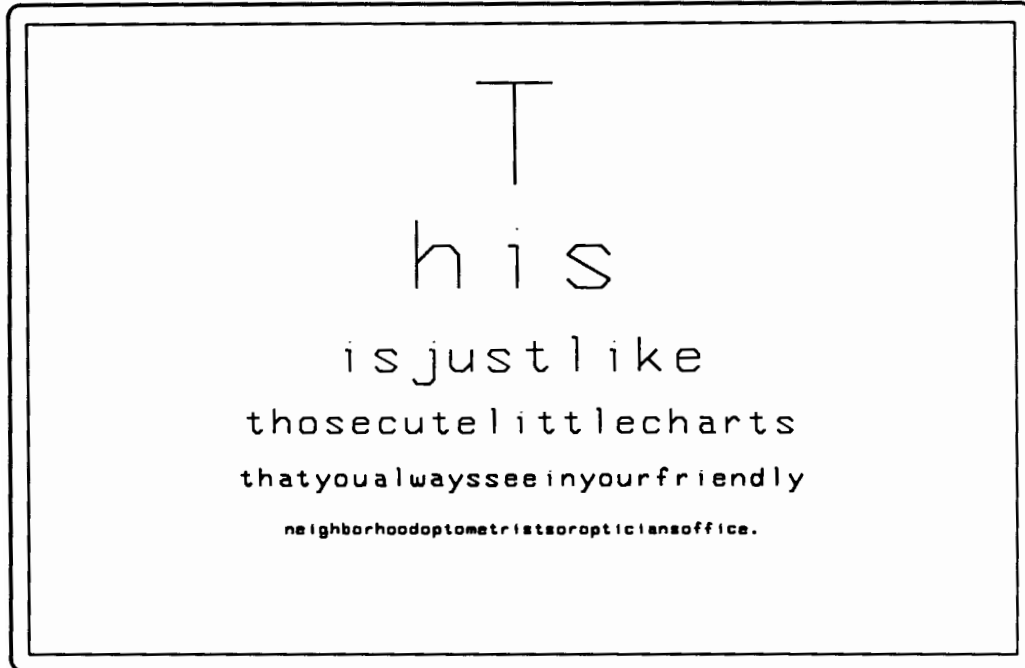
When the second WINDOW command is executed in line 50, the graphics system attempts to convert the large REAL coordinate values of the graphics pen position to INTEGERS according to the newly scaled plotting area. Since the original plotting area is much greater than the newly scaled plotting area, these large REAL coordinate values cannot be mapped into INTEGER values (since INTEGERS are limited to the range 0 through 65,535), and a math overflow error occurs.

More on Labeling a Plot

Three statements complement the LABEL statement. 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. The amount of blank space depends on which character is contained in the cell.

This small program shows how the CSIZE statement changes the size of characters. (Run the example program in file Csize.)

```
100 CLEAR SCREEN           ! Clear the alpha display
110 DIM Text$(50)         ! Allow the long strings
120 GINIT                  ! Initialize various graphics parameters
130 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
140 GRAPHICS ON           ! Turn on the graphics screen
150 FRAME                  ! Draw a box around the screen
160 WINDOW -1,1,10,1     ! Anisotropic units
170 LOG 4                  ! Bottom center of labels is ref. pt.
180 FOR I=1 TO 6          ! Six labels total
190   READ Csize,Text$    ! Read the characters cell size and text
200   CSIZE Csize        ! Use Csize
210   MOVE 0,SQR(I)*3+1  ! Move to appropriate place
220   LABEL Text$        ! Write the text
230 NEXT I                ! Looplooplooplooplooploop
240 DATA 30,T,20,his,10,isjustlike,7,thosecutelittlecharts
250 DATA 5,thatyoualwaysseeinyourfriendly
260 DATA 3,neighborhoodoptometristsoropticiansoffice.
270 END
```



Changing Graphics Character Size

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 that the WINDOW statement 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. (Run the example program in file CharCell.)

```

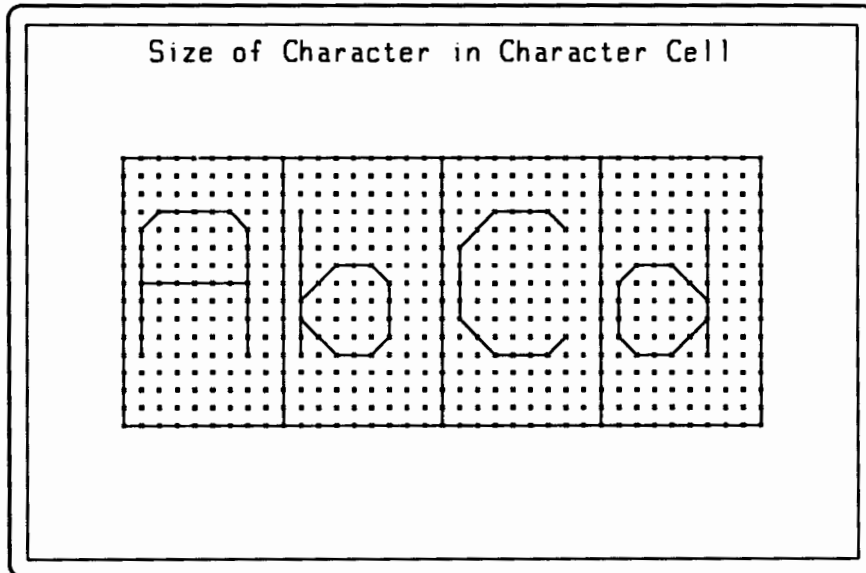
100 CLEAR SCREEN           ! Clear the alpha display
110 GINIT                 ! Initialize various graphics parameters
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON           ! Turn on the graphics screen
140 FRAME                 ! Draw a box around the screen
150 SHOW 0,36,-7.5,22.5   ! Isotropic units; Left/Right/Bottom/Top
160 FOR X=0 TO 36         ! \

```

```

170   FOR Y=0 TO 15           ! \
180     MOVE X-.1,Y+.1       ! \
190     DRAW X+.1,Y-.1       ! \
200     MOVE X+.1,Y+.1       ! > Draw all the little Xs.
210     DRAW X-.1,Y-.1       ! /
220   NEXT Y                 ! /
230 NEXT X                   ! /
240   FOR I=0 TO 27 STEP 9    ! \
250     CLIP I,I+9,0,15      ! \
260     FRAME                 ! > Draw boxes around the character cells
270   NEXT I                 ! /
280   CLIP OFF               ! Deactivate clipping so LABELs will work
290   CSIZE 50               ! Character cells half the screen high
300   MOVE 0,0               ! Starting point (LORG 1 by default)
310   LABEL "AbCd"           ! Sample letters
320   CSIZE 7,.45           ! \
330   LORG 6                 ! \
340   MOVE 18,22            ! > Write the title
350   LABEL "Size of Character in Character Cell" ! /
360   END

```



Character Cells (CharCell)

As the diagram shows, a character is drawn inside a rectangle, with some space on all four sides. The character's height and width are measured in GDUs and are specified by the CSIZE statement. Program lines 250 through 280 subdivide the rectangle into four 9 wide by 15 high grids. 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.

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 0.5.

CSIZE 3 Cell 3 GDUs high, aspect ratio 0.6 (default).

CSIZE 6, .3 Cell 6 GDUs high, aspect ratio 0.3 (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 since it 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. (Run the example program in file Lorg.)

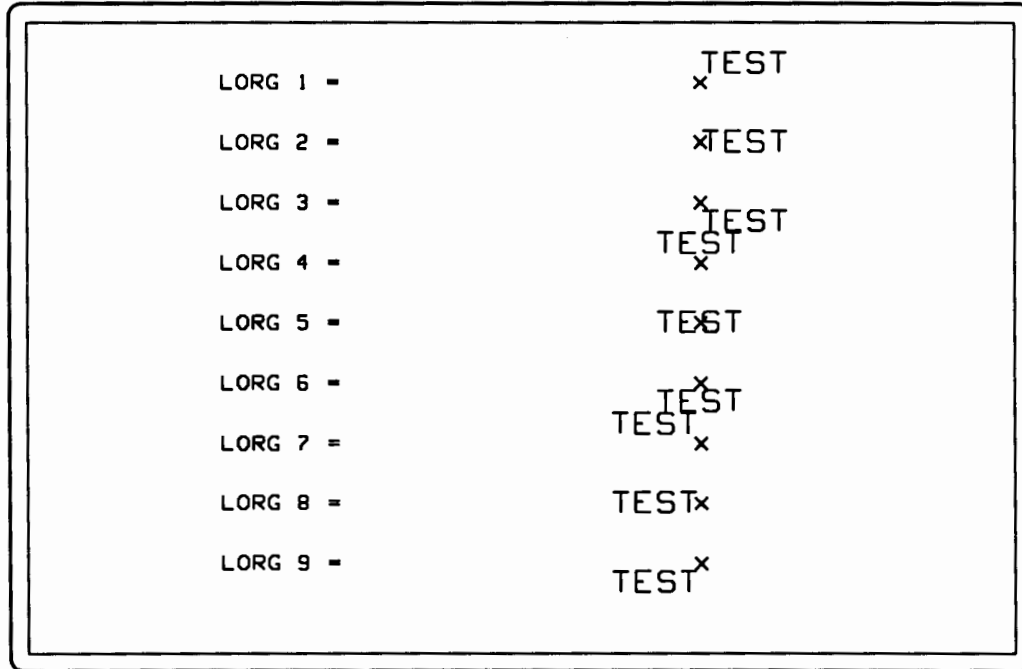
```

100 CLEAR SCREEN            ! Clear the alpha screen
110 GINIT                    ! Initialize various graphics parameters
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON             ! Turn on the graphics screen
140 SHOW 0,10,10.5,0        ! Isotropic scaling: Left/Right/Bottom/Top
150 FRAME                    ! Draw a box around the screen
160 FOR Lorg=1 TO 9         ! Loop on LORG parameters
170    LORG 2                ! Left-center origin for the "LORG n ="
180    CSIZE 4               ! Characters cell 4 GDUs high
190    MOVE 0,Lorg            ! Move to position for "LORG n =" label
200    LABEL "LORG";Lorg;"=" ! Write the label
210    MOVE 8+.1,Lorg+.1     ! \
220    DRAW 8-.1,Lorg-.1     ! \
230    MOVE 8-.1,Lorg+.1     ! > Draw an "X" to show where pen is
240    DRAW 8+.1,Lorg-.1     ! /
250    LORG Lorg             ! Specify LORG for "TEST",
260    CSIZE 6               ! ... and larger letters
270    MOVE 8,Lorg            ! Move the center of the "X"
280    LABEL "TEST"         ! Write "TEST", using current LORG

```

290 NEXT Lorg
300 END

! And so forth



Label Origins (Lorg)

The x's indicate where the pen was moved to before labeling the word "TEST". This diagram means, for example, that 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. LDIR 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:

8-24 Graphics Techniques

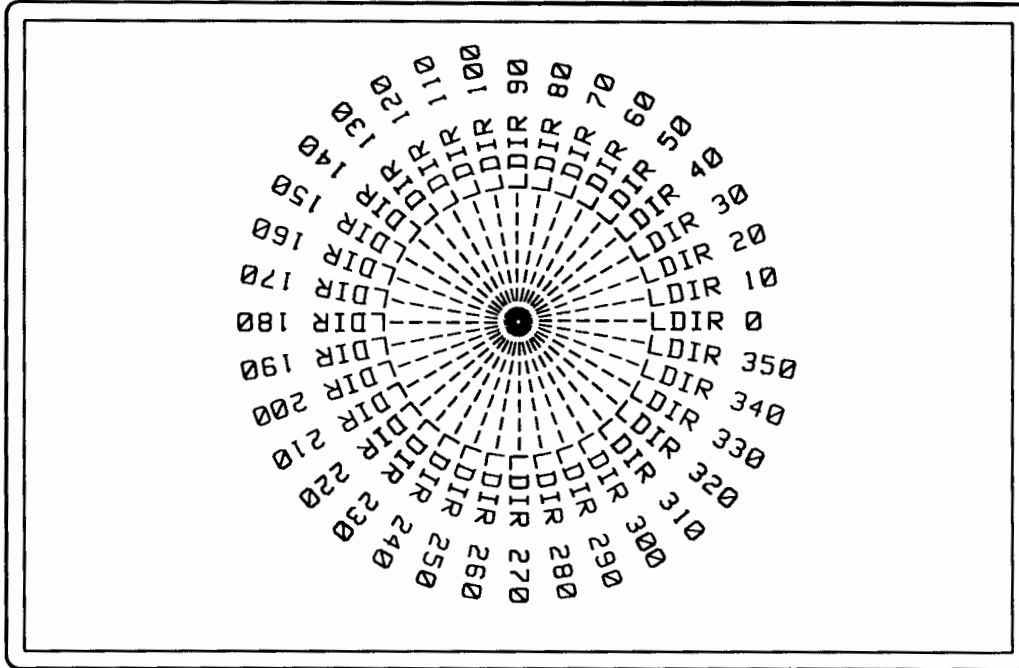
- LDIR 0 Writes label horizontally to the right.
- LDIR 90 Writes label vertically, ascending.
- LDIR 14 Writes label ascending a gentle slope, up and right.
- LDIR 180 Writes label upside down.
- LDIR 270 Writes label vertically, descending.

In the program below, note that LORG 2 was specified and remained in effect for many LDIRs. Each label is centered on the left edge relative to the *label*. (Run the example program in file Ldir.)

```

100 CLEAR SCREEN                   ! Clear the alpha display
110 GINIT                           ! Initialize various graphics parameters
120 PLOTTER IS CRT,"INTERNAL"       ! Use the internal screen
130 GRAPHICS ON                     ! Turn on the graphics screen
140                                 ! (Series 200 computers)
150 FRAME                           ! Draw a box around the screen
160 WINDOW -1,1,-1.1,1             ! Anisotropic units; Left/Right/Bottom/Top
170 DEG                             ! Angular mode: Degrees
180 LORG 2                          ! Label origin is left center
190 FOR Angle=0 TO 350 STEP 10      ! Every 10 degrees
200     LDIR Angle                   ! Labelling angle
210     MOVE 0,0                    ! Move to center of screen
220     LABEL "-----LDIR";Angle   ! Write using the current LDIR
230 NEXT Angle                      ! And so on
240 END

```

Changing Label Directions (Ldir)

The label origin specified by LORG is relative to the label, not the plotting surface, and is independent of the current label direction. For example, if you specified

```

LORG 3
DEG
LDIR 90
MOVE 6,8

```

8

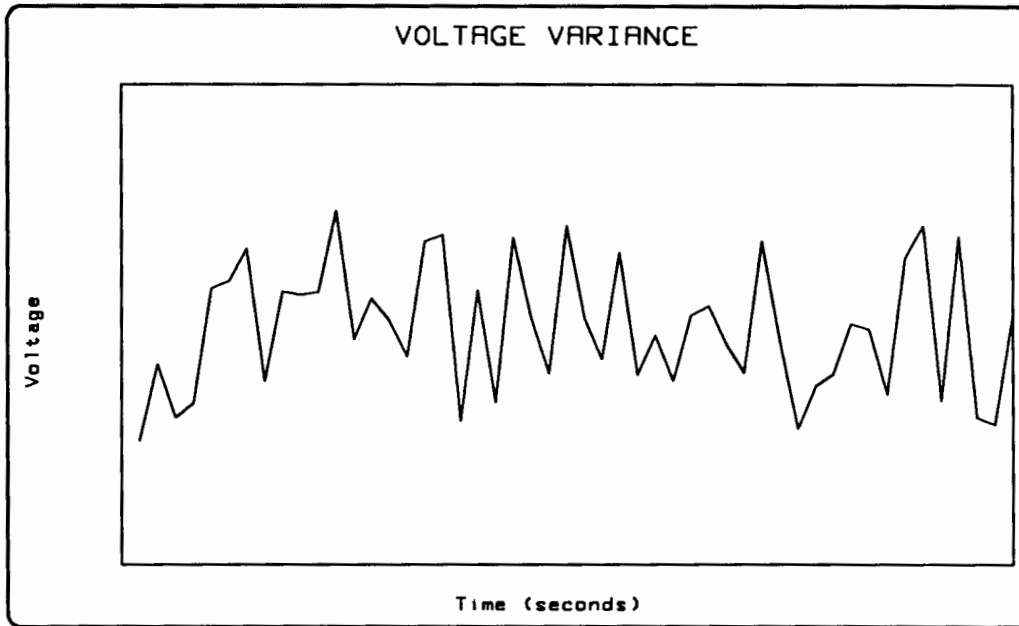
and then wrote the label, it would be written going straight up, not horizontally. Therefore, it is the upper left corner of the label which is at point 6,8 *relative to the rotated label*. Relative to the plotting device, however, it is the lower left corner of the label which is at 6,8 (in this example) because the label has been rotated.

The statement which actually causes labels to be written is LABEL. LABEL accounts for the most recently specified CSIZE, LDIR and LORG when it

writes a label. You must position the label, however, by using (for example) a MOVE statement to get to the point at which you want the label to be placed.

All four statements have been utilized in the following program. (Run the example program in file SinLabel.)

```
100 CLEAR SCREEN           ! Clear the alpha display
110 GINIT                 ! Initialize various graphics parameters.
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON          ! Turn on the graphics screen
140 X_gdu_max=100*MAX(1,RATIO) ! Determine how many GDUs wide the screen is
150 Y_gdu_max=100*MAX(1,1/RATIO) ! Determine how many GDUs high the screen is
160 LORG 6                ! Reference point: center of top of label
170 MOVE X_gdu_max/2,Y_gdu_max ! Move to middle of top of screen
180 LABEL "VOLTAGE VARIANCE" ! Write title of plot
190 DEG                  ! Angular mode is degrees (used in LDIR)
200 LDIR 90              ! Specify vertical labels
210 CSIZE 3.5           ! Specify smaller characters
220 MOVE 0,Y_gdu_max/2  ! Move to center of left edge of screen
230 LABEL "Voltage"     ! Write Y-axis label
240 LORG 4               ! Reference point: center of bottom of label
250 LDIR 0               ! Horizontal labels again
260 MOVE X_gdu_max/2,.07*Y_gdu_max ! X: center of screen; Y: above key labels
270 LABEL "Time (seconds)" ! Write X-axis label
280 VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
                        ! Define subset of screen area
290 FRAME                ! Draw a box around defined subset
300 WINDOW 0,100,16,18  ! Anisotropic scaling: left/right/bottom/top
310 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
320   PLOT X,RND*16.5    ! Get a data point and plot it against X
330 NEXT X               ! et cetera
340 END
```



Example of Labeling (SinLabel)

Many times it's nice to have **bold** letters. You can achieve this effect by plotting the label several times, moving the label origin just slightly each time. In the following version of the program, notice lines 180 through 210. The loop variable, I, goes from $-.3$ to $.3$ by tenths. This is the offset in the X direction of the label origin. Since this is being labeled with LORG 6 in effect, the label origin (the point moved to immediately prior to labeling) represents the center of the top edge of the label. (Run the example program in file SinLabel2.)

8

```

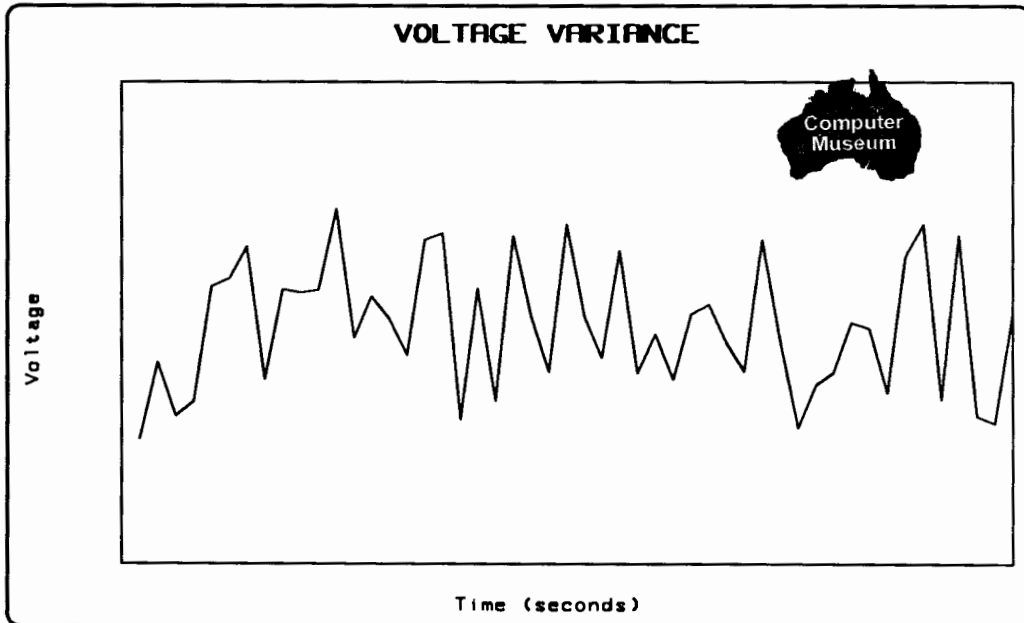
100 CLEAR SCREEN           ! Clear the alpha display
110 GINIT                  ! Initialize various graphics parameters.
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON           ! Turn on the graphics screen
140 X_gdu_max=100*MAX(1,RATIO) ! Determine how many GDUs wide the screen is
150 Y_gdu_max=100*MAX(1,1/RATIO) ! Determine how many GDUs high the screen is
160 LORG 6                 ! Reference point: center of top of label
170 FOR I=-.3 TO .3 STEP .1 ! Offset of X from starting point
180     MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
190     LABEL "VOLTAGE VARIANCE" ! Write title of plot

```

```

200 NEXT I                ! Next position for title
210 DEG                  ! Angular mode is degrees (used in LDIR)
220 LDIR 90              ! Specify vertical labels
230 CSIZE 3.5            ! Specify smaller characters
240 MOVE 0,Y_gdu_max/2   ! Move to center of left edge of screen
250 LABEL "Voltage"      ! Write Y-axis label
260 LORG 4                ! Reference point: center of bottom of label
270 LDIR 0                ! Horizontal labels again
280 MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
290 LABEL "Time (seconds)" ! Write X-axis label
300 VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
                        ! Define subset of screen area
310 FRAME                ! Draw a box around defined subset
320 WINDOW 0,100,16,18   ! Anisotropic scaling: left/right/bottom/top
330 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
340   PLOT X,RND+16.5     ! Get a data point and plot it against X
350 NEXT X                ! et cetera
360 END

```



Bold Labels (SinLabel2)

This method can also be used to offset in the Y direction. Or, offset both X and Y. This will give you characters that are thick in a diagonal direction, which makes them look like they are coming out of the page at you. However, a more typical bolding is produced by offsetting only in the X direction.

Axes and Grids

AXES and GRID do similar operations. We've already seen how to use the AXES statement. The GRID statement causes the major tick marks to extend all the way across the plotting surface.

Once we have the axes drawn, we must label various points along them with numbers designating the values at those points. Once again, we use the LABEL statement. (Run the example program in file SinAxes.)

```

100 CLEAR SCREEN           ! Clear the alpha display
110 GINIT                  ! Initialize various graphics parameters.
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON           ! Turn on the graphics screen
140 X_gdu_max=100*MAX(1,RATIO) ! Determine how many GDUs wide the screen is
150 Y_gdu_max=100*MAX(1,1/RATIO) ! Determine how many GDUs high the screen is
160 LORG 6                 ! Reference point: center of top of label
170 FOR I=-.3 TO .3 STEP .1 ! Offset of X from starting point
180   MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
190   LABEL "VOLTAGE VARIANCE" ! Write title of plot
200 NEXT I                 ! Next position for title
210 DEG                    ! Angular mode is degrees (used in LDIR)
220 LDIR 90                ! Specify vertical labels
230 CSIZE 3.5              ! Specify smaller characters
240 MOVE 0,Y_gdu_max/2     ! Move to center of left edge of screen
250 LABEL "Voltage"        ! Write Y-axis label
260 LORG 4                 ! Reference point: center of bottom of label
270 LDIR 0                 ! Horizontal labels again
280 MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
290 LABEL "Time (seconds)" ! Write X-axis label
300 VIEWPORT .1*X_gdu_max,.98*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
      ! Define subset of screen area
310 FRAME                  ! Draw a box around defined subset
320 WINDOW 0,100,16,18    ! Anisotropic scaling: left/right/bottom/top
330 AXES 1,.05,0,16,10,5,3 ! Draw axes with appropriate ticks
340 CLIP OFF               ! So labels can be outside VIEWPORT limits
350 CSIZE 2.5,.5          ! Smaller chars for axis labeling
360 LORG 6                 ! Ref. pt: Top center  \

```

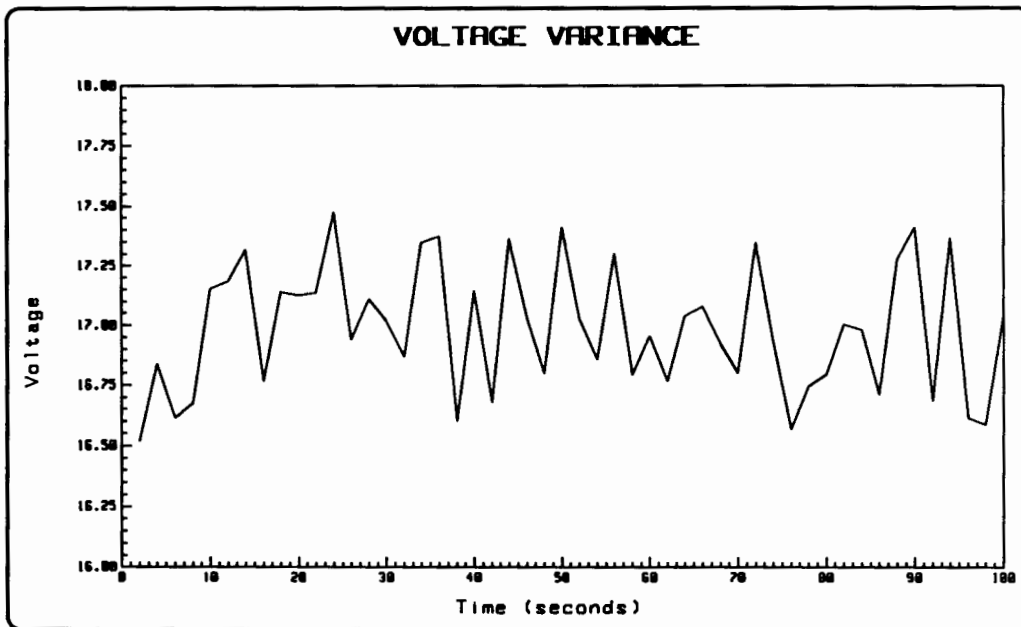
8

8-30 Graphics Techniques

```

370  FOR I=0 TO 100 STEP 10      ! Every 10 units      | \
380    MOVE I,15.99             ! A smidgeon below X-axis | > Label X-axis
390    LABEL USING "#,K";I      ! Compact; no CR/LF     | /
400  NEXT I                     ! et sequens           | /
410  LORG 8                     ! Ref. pt: Right center  | \
420  FOR I=16 TO 18 STEP .25    ! Every quarter         | \
430    MOVE -.5,I              ! Smidgeon left of Y-axis | > Label Y-axis
440    LABEL USING "#,DD.DD";I  ! DD.DD; no CR/LF      | /
450  NEXT I                     ! et sequens           | /
460  PENUP
470  FOR X=2 TO 100 STEP 2      ! Points to be plotted
480    PLOT X,RND+16.5         ! Plot a data point
490  NEXT X
500  END

```



Labeled Axes (SinAxes)

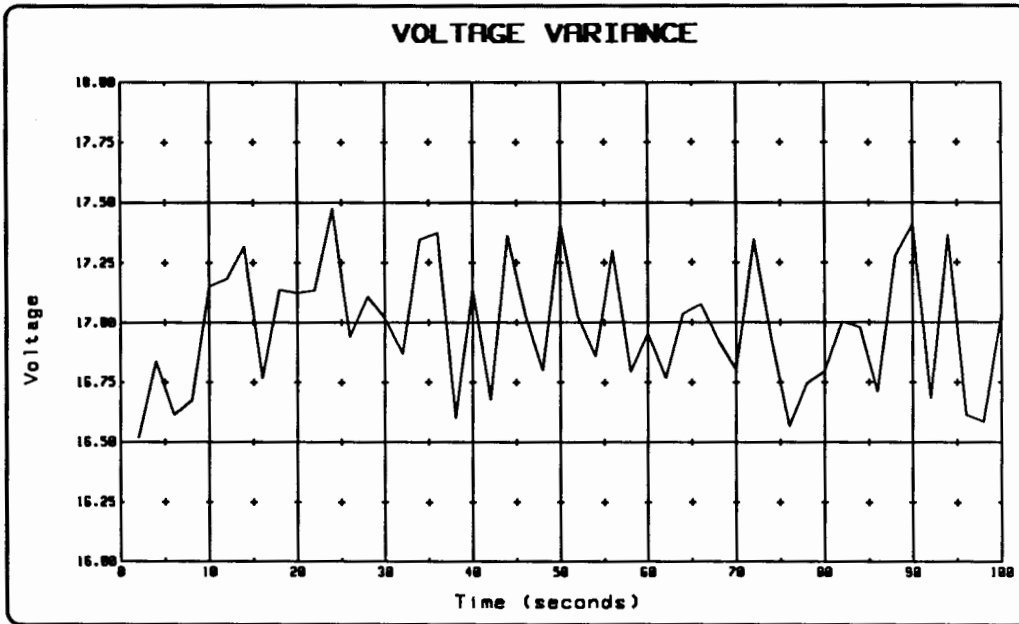
Note that the tick marks drawn by the AXES statement extend only toward the interior of the graph. Clipping (automatically put into effect by the VIEWPORT statement) was still active at the soft clip limits. If the CLIP OFF statement had been placed *before* the AXES statement, the tick marks

would have extended on both sides of the axes. However, the axes themselves would have extended across the entire width of the hard clip limits and right through the axes' labels.

The CLIP OFF statement was necessary, though. The LABEL statement draws the letters as a series of vectors (lines), and any lines which are outside the current soft clip limits (when CLIP is ON) are cut off. This means that had line 350 (the CLIP OFF) been missing from the program, none of the axis labels would have been drawn, since they are *all* outside the VIEWPORT area. Of course, the main titles ("VOLTAGE VARIANCE", "Voltage", and "Time (seconds)") would still have been drawn, because they are done *before* the VIEWPORT is executed.

If your graph needs to be read with more precision than the AXES statement affords, you can use the GRID statement. This is similar to AXES, except the major ticks extend across the entire soft clip area, and the minor ticks for X and Y intersect in little crosses between the grid lines. The previous program has only one change: the AXES statement has been replaced by a GRID statement.

```
GRID 5,.25,0,16,2,2,1      ! Draw grid with appropriate ticks
```



Labeled Grid

Note that not only was AXES replaced by GRID, some of the parameters were changed also because the minor ticks specified in the AXES statement were so close together that the minor tick crosses drawn by the GRID statement would have overlapped. The end result would have been a grid with even the minor ticks extending all the way across the soft clip area.

Strategy: Axes Versus Grids

On many occasions, an application is defined such that there is no question as to which statement to use. Other times, however, you want to weigh the alternatives carefully before setting your program in concrete. To aid you in the decision, here are some advantages and disadvantages to both statements.

Advantages of AXES:

- It executes faster than GRID for two reasons. First, the computer does less calculating, and second, the computer does less actual drawing of lines. This

becomes especially evident when sending a plot to a hard-copy plotting device where a physical pen must be hauled around.

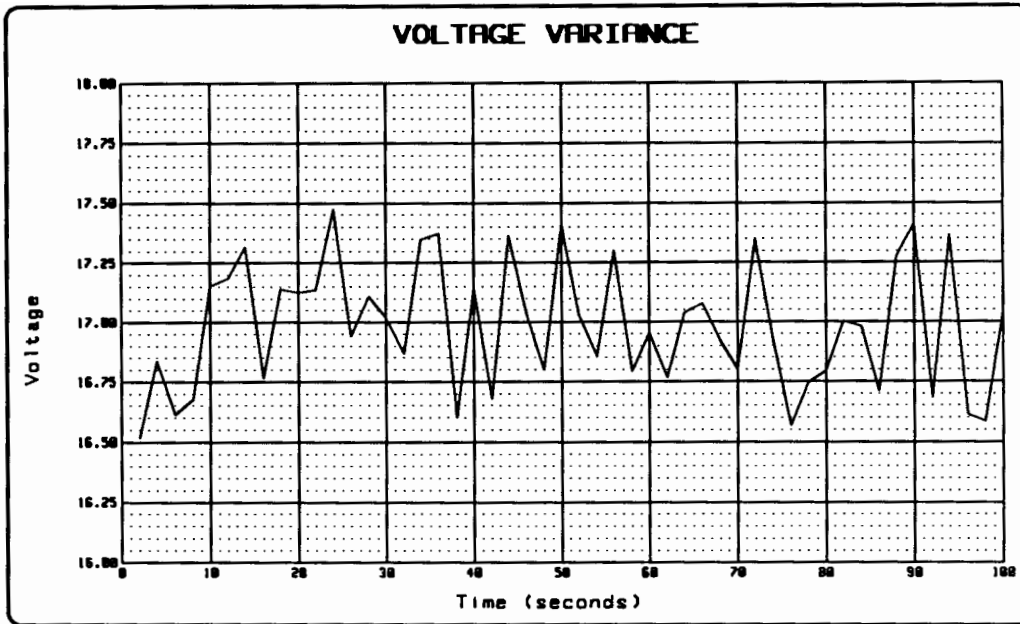
- It does not clutter the plot as much. Reference points are available at the axes, but there is no question about where the data curve is. When using GRID, it is possible to lose the data curve among the reference lines if it is close to being horizontal or vertical.

Advantages of GRID:

- Interpolation and estimation are more accurate due to the great number of reference ticks and lines; one need not estimate horizontal and vertical lines to refer back to the axis labels.
- Usually there is no need to use a FRAME statement to completely enclose the soft clip limits, as is often desired, because the major tick marks from the GRID statement would probably redraw the lines. Of course, this depends on the major tick count.

However, if you want to estimate data points accurately from the finished plot, but also want to prevent the plot from appearing cluttered, it can be done. Below is a plot drawn by a program identical to the previous one except for the GRID statement. The GRID statement used specifies exactly the same parameters as the AXES statement (two programs ago) with one exception: the major tick length parameter is reduced. This causes the tick crosses to be reduced to dots. Using this strategy allows easy interpolation of data points (to the same accuracy previously used in the AXES statement), but does not clutter the graph as much as normal ticks would.

```
GRID 1,.05,0,16,10,5,.0001    ! Draw grid with appropriate ticks
```



Labeled Grid with More Tick Marks

Be aware that the computer still thinks of these smaller tick marks as crosses, which means that both the horizontal and vertical components must be drawn. This looks to you like drawing and then redrawing each dot. Therefore, you can expect this type of grid to take a while to plot when you send it to a hard-copy plotter.

Another way to compromise between ease of interpolation and lack of clutter is to use *both* AXES and GRID in the same program. Note the program below. GRID is used for the major tick lines, but since the minor tick crosses are not desired within each rectangle between the major tick lines, AXES is used to specify minor ticks. (Run the example program in file SinGrdAxes.)

8

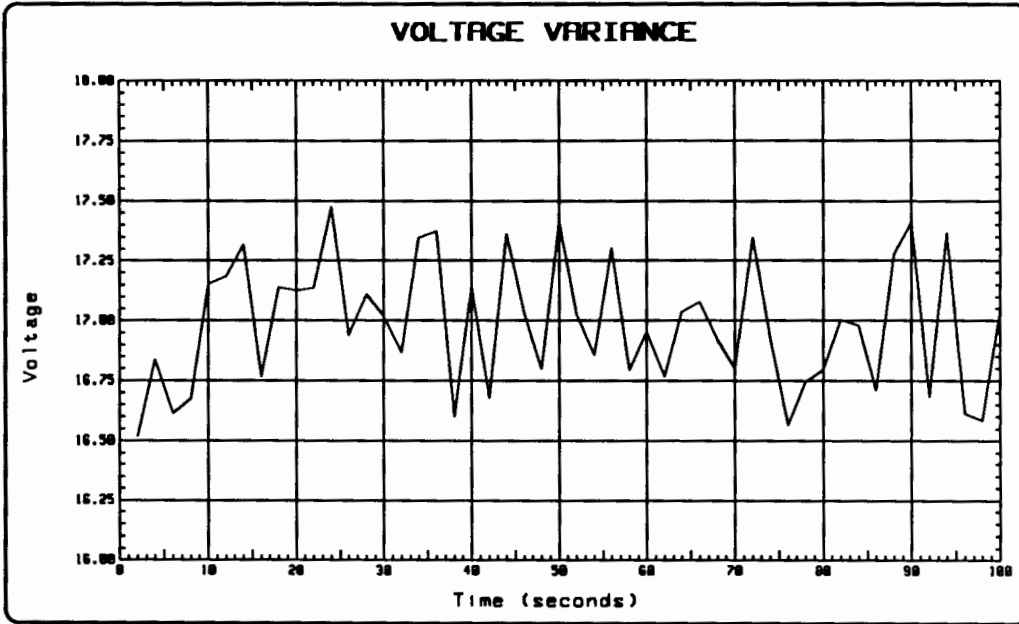


```

100 CLEAR SCREEN          ! Clear the alpha display
110 GINIT                 ! Initialize various graphics parameters.
120 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
130 GRAPHICS ON           ! Turn on the graphics screen
140 LORG 6                ! Reference point: center of top of label
150 X_gdu_max=100*MAX(1,RATIO) ! Determine how many GDUs wide the screen is
160 Y_gdu_max=100*MAX(1,1/RATIO) ! Determine how many GDUs high the screen is
170 FOR I=-.3 TO .3 STEP .1 ! Offset of X from starting point
180   MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
190   LABEL "VOLTAGE VARIANCE" ! Write title of plot
200 NEXT I                ! Next position for title
210 DEG                   ! Angular mode is degrees (used in LDIR)
220 LDIR 90               ! Specify vertical labels
230 CSIZE 3.5             ! Specify smaller characters
240 MOVE 0,Y_gdu_max/2    ! Move to center of left edge of screen
250 LABEL "Voltage"       ! Write Y-axis label
260 LORG 4                ! Reference point: center of bottom of label
270 LDIR 0                ! Horizontal labels again
280 MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
290 LABEL "Time (seconds)" ! Write X-axis label
300 VIEWPORT .1*X_gdu_max,.98*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
    ! Define subset of screen area
310 WINDOW 0,100,16,18    ! Anisotropic scaling: left/right/bottom/top
320 AXES 1,.05,0,16,5,5,3 ! Draw axes intersecting at lower left
330 AXES 1,.05,100,18,5,5,3 ! Draw axes intersecting at upper right
340 GRID 10,.25,0,16,1,1 ! Draw grid with no minor ticks
350 CLIP OFF              ! So labels can be outside VIEWPORT limits
360 CSIZE 2.5,.5         ! Smaller chars for axis labeling
370 LORG 6                ! Ref. pt: Top center      |\
380 FOR I=0 TO 100 STEP 10 ! Every 10 units          | \
390   MOVE I,15.99         ! A smidgeon below X-axis | > Label X-axis
400   LABEL USING "#,K";I ! Compact; no CR/LF      | /
410 NEXT I                ! et sequens             | /
420 LORG 8                ! Ref. pt: Right center  |\
430 FOR I=16 TO 18 STEP .25 ! Every quarter          | \
440   MOVE -.5,I          ! Smidgeon left of Y-axis | > Label Y-axis
450   LABEL USING "#,DD.DD";I ! DD.D; no CR/LF        | /
460 NEXT I                ! et sequens             | /
470 PENUP                 ! LABEL statement leaves the pen down
480 FOR X=2 TO 100 STEP 2 ! Points to be plotted ...
490   PLOT X,RND+16.5     ! Get a data point and plot it against X
500 NEXT X                ! et cetera
510 END

```

8



Using AXES and GRID (SinGrdAxes)

Note that *two* AXES statements were used. The parameters are identical save for the position of the intersection. The first AXES specifies an intersection position of 0,16: the lower left corner of the soft clip area. The second specifies an intersection position of 100,18: the upper right corner of the soft clip area.

Also note that the FRAME statement was removed; the lines around the soft clip limit were being drawn by the AXES statements and the GRID statement anyway.

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; the pen cannot 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
```

and it can be turned back on, using the same limits, by

```
CLIP ON
```

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, three different drawing modes are available. (For selecting pens with a color CRT, see "Color Graphics".) The three pens perform the following actions:

Monochromatic Pens

Pen Number	Function
1	Draws lines (turns on pixels)
-1	Erases lines (turns off pixels)
0	Complements lines (changes pixels' states)

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 of the lines.

This concept is illustrated by the following program. However, since it is a dynamic display, no figure is given. Line 150 of the program defines the type of operation the program will exhibit. If Pen equals zero, all lines will complement, because lines 610 and 680 select pen -0 and +0, which are identical. When you wish to change the program to drawing and erasing mode, change line 150 to say Pen=1. Then lines 610 and 680 will select pens -1 and +1, respectively. (Run the example program in file Pen.)

```
100 CLEAR SCREEN ! Clear the alpha display
110 INTEGER Polygon,Polygons,Side,Sides,Pen ! Make loops faster
120 Polygons=20 ! How many polygons?
130 Sides=3 ! How many sides apiece?
140 Pen=0 ! 1: Draw/erase; 0: Complement
150 ALLOCATE INTEGER X(0:Polygons-1,1:Sides),Y(0:Polygons-1,1:Sides)
160 ALLOCATE INTEGER Dx(Sides),Dy(Sides)
170 RANDOMIZE ! Different each time
180 GINIT ! Initialize graphics parameters
190 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
200 GRAPHICS ON ! Turn on graphics screen
210 WINDOW 0,511,0,389 ! Integer arithmetic is faster
220 PEN Pen ! Select appropriate pen
```

```

230 FOR Side=1 TO Sides                ! For each vertex ...
240   X(0,Side)=RND*512                ! ... define a starting point ...
250   Y(0,Side)=RND*390                ! ... for both X and Y ...
260   PLOT X(0,Side),Y(0,Side)        ! ... then draw to that point.
270 NEXT Side                          ! et cetera
280 IF Sides>2 THEN PLOT X(0,1),Y(0,1) ! If simple line, don't close
290 GOSUB Define_deltas                ! Get dx and dy for each vertex
300 FOR Polygon=1 TO Polygons-1       ! Draw all the polygons
310   PENUP                             ! Don't connect polygons
320   FOR Side=1 TO Sides               ! Each vertex of each polygon
330     Temp=X(Polygon-1,Side)+Dx(Side) ! Avoid recalculation
340     IF Temp>511 THEN                ! \
350       Dx(Side)=-Dx(Side)           ! \
360     ELSE ! (it's not off right side) ! > Is X out of range?
370       IF Temp<0 THEN Dx(Side)=-Dx(Side)! /
380     END IF ! (off right side?)      ! /
390     X(Polygon,Side)=X(Polygon-1,Side)+Dx(Side) ! Calculate next X
400     Temp=Y(Polygon-1,Side)+Dy(Side) ! Avoid recalculation
410     IF Temp>389 THEN                ! \
420       Dy(Side)=-Dy(Side)           ! \
430     ELSE ! (it's not off top)       ! > Is Y out of range?
440       IF Temp<0 THEN Dy(Side)=-Dy(Side)! /
450     END IF ! (off the top?)         ! /
460     Y(Polygon,Side)=Y(Polygon-1,Side)+Dy(Side) ! Calculate new Y
470     PLOT X(Polygon,Side),Y(Polygon,Side)! Draw line to new point
480   NEXT Side                        ! Loop for next side of polygon
490   IF Sides>2 THEN PLOT X(Polygon,1),Y(Polygon,1)! If line, don't close
500 NEXT Polygon                       ! Get each polygon
510 New=0                               ! Start re-use at entry 0
520 ON CYCLE 10 GOSUB Define_deltas    ! Change deltas periodically
530 LOOP                                ! Ad infinitum ...
540 IF New=0 THEN                       ! Boundary condition?
550   Previous=Polygons-1               ! Start re-using over
560 ELSE ! (new>0)
570   Previous=(Previous+1) MOD Polygons ! Re-use next entry
580 END IF ! (new=0?)
590 PENUP                               ! Don't connect polygons
600 PEN -Pen                            ! This works either way for Pen
610 DISABLE                             ! Don't interrupt in "Side" loop
620 FOR Side=1 TO Sides                 ! \
630   PLOT X(New,Side),Y(New,Side)      ! \
640 NEXT Side                           ! > Erase oldest line
650 IF Sides>2 THEN PLOT X(New,1),Y(New,1)! /
660 PENUP                               ! /

```

```

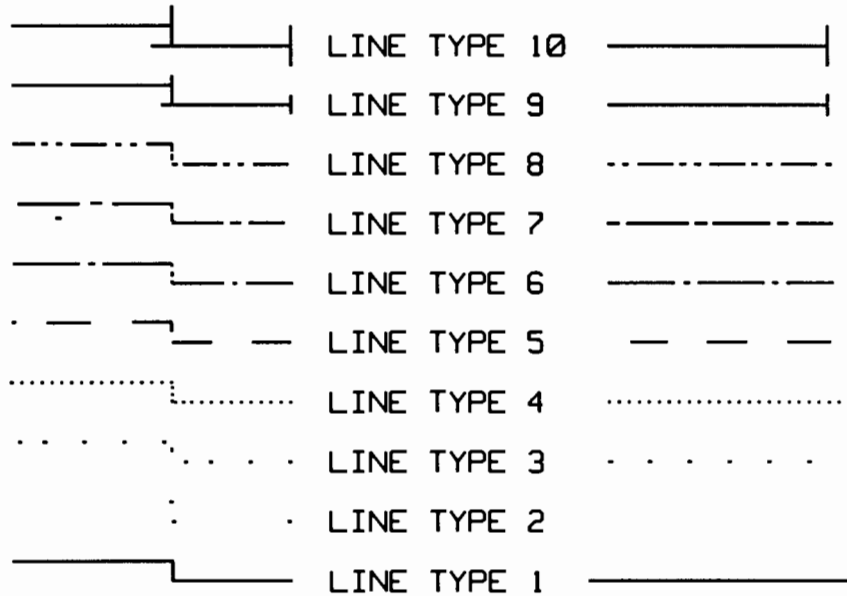
670   PEN Pen                                ! Drawing pen
680   FOR Side=1 TO Sides                    ! \
690     Temp=X(Previous,Side)+Dx(Side)      ! /
700     IF Temp>511 THEN                    ! /
710       Dx(Side)=-Dx(Side)               ! /
720     ELSE                                ! /
730       IF Temp<0 THEN Dx(Side)=-Dx(Side)! /
740     END IF                              ! /
750     X(New,Side)=X(Previous,Side)+Dx(Side) ! /
760     Temp=Y(Previous,Side)+Dy(Side)      ! /
770     IF Temp>389 THEN                    ! / Draw the new line
780       Dy(Side)=-Dy(Side)               ! / same way as before.
790     ELSE                                ! /
800       IF Temp<0 THEN Dy(Side)=-Dy(Side)! /
810     END IF                              ! /
820     Y(New,Side)=Y(Previous,Side)+Dy(Side) ! /
830     PLOT X(New,Side),Y(New,Side)       ! /
840   NEXT Side                             ! /
850   IF Sides>2 THEN PLOT X(New,1),Y(New,1)! /
860   ENABLE                                ! Interrupts OK again
870   New=(New+1) MOD Polygons              ! Next one to re-use.
880 END LOOP                                ! End of infinite loop
890 Define_deltas: ! -----
900   FOR Side=1 TO Sides                    ! For each vertex
910     Dx(Side)=RND*3+2                     ! Magnitude of this dx
920     IF RND<.5 THEN Dx(Side)=-Dx(Side)   ! Sign of this dx
930     Dy(Side)=RND*3+2                     ! Magnitude of this dy
940     IF RND<.5 THEN Dy(Side)=-Dy(Side)   ! Sign of this dy
950   NEXT Side                             ! et cetera
960   RETURN                                 ! back to the main program
970 END

```

When running the program in complementing mode, observe that a pixel is on only if it has been acted upon by an odd number of line segments.

Selecting Line Types

When a graph attempts to convey several different kinds of information, colors are often used: The red curve signifies one thing, the blue curve signifies another thing, and so on. But when only one color is available, as on a monochromatic CRT, this method cannot be used. Something that *can* be used, however, is different line types. There are ten line types available:



Line Types

As you can see, LINE TYPE 1 draws a solid line. LINE TYPE 2 draws only the end points of the lines and is the same as moving to a new point, dropping the pen, lifting the pen, and repeating. LINE TYPEs 3 through 8 are patterned sequences of on and off. With these, the length of each pattern; i.e., the distance the line extends before the on/off pattern begins to repeat, can be specified by supplying a second parameter in the LINE TYPE statement. This second parameter specifies distance in GDUs.

For example,

`LINE TYPE 5,15`

8 tells the computer to start using a simple dashed line, and to proceed a total of 15 GDUs before starting the pattern over. On the CRT, the repeat length will be rounded to a multiple of five, with a minimum value of five.

LINE TYPEs 9 and 10 are solid lines with a minor and major tick mark at the end of each line, respectively. The tick mark will be either horizontal or vertical. The orientation of the tick marks will be whatever is farther from the angle of the line just drawn. For example, if you draw a line at a 30° angle, it

is closer to being horizontal than it is to being vertical. Thus, the tick mark at the end of the line will be vertical. The value for major tick size is 2 GDUs, and minor tick length is one half the major tick length.

For all line types, the computer remembers where in the pattern a line segment ended. Therefore, when you start drawing another line segment, the line pattern will continue from where it left off. If you want the pattern to start over, just re-execute the LINE TYPE statement.

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 on a file. This can be done with the GSTORE command. First, you must have an INTEGER array of sufficient size to hold all the data in the graphics raster. The array size varies depending on what computer system you have in general and what monitor you have in particular. A formula for calculating array size is:

$$\left(\frac{\text{number X pixels} * \text{number Y pixels}}{16} \right) \text{ number of bits per pixel}$$

A monochromatic display has 1 bit per pixel. The Model 236C color computer has 4 bits per pixel. The Series 300 color monitors have either 4, 6 or 8 bits per pixel. But rather than getting intimately involved with screen resolution and the number of bits per pixel, there is a shortcut. The fifth and sixth elements of the integer array passed back by GESCAPE operation selector 3 specify the number of rows and columns an integer array must have to contain the entire graphics image.

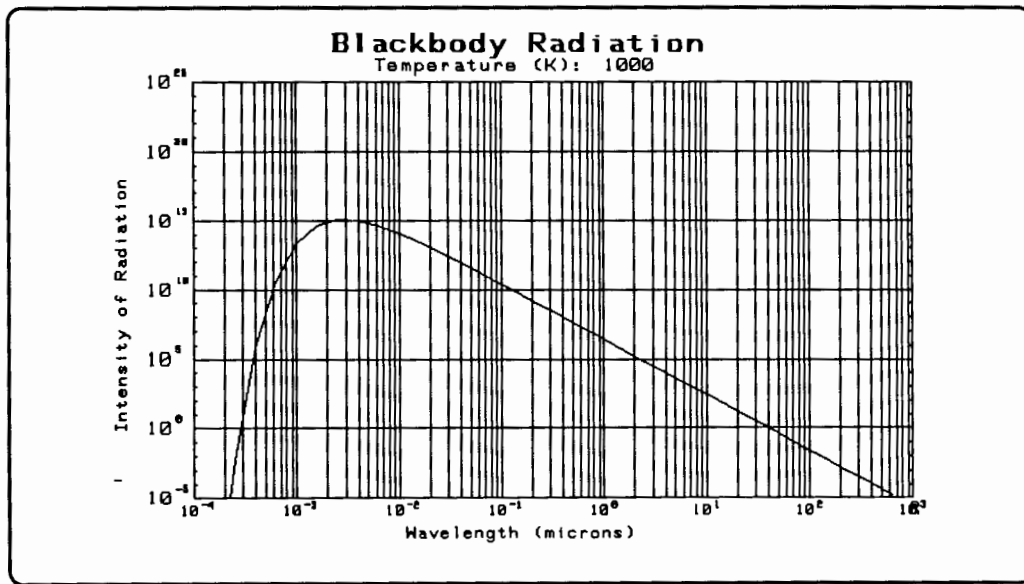
For example:

```
...
100  GINIT
110  INTEGER A(1:6)
120  GESCAPE CRT,3;A(*)
130  PRINT "Rows required for array:  ";A(5)
140  PRINT "Columns required for array: ";A(6)
150  ALLOCATE INTEGER Gscreen(A(5),A(6))
160  GSTORE Gscreen(*)
170  END
...
```

The array `Gscreen` is allocated the size specified by the "rows" and "columns" numbers from the `GESCAPE` return array. This array holds the picture itself and doesn't care how the information got to the screen, or in what order the different parts of the picture were produced.

In the `Gstore` program, the image is drawn with normal plotting commands, and then, *after the fact*, the image is read from the graphics area in memory, and placed into the array. After the array is filled by the `GSTORE`, a curve is plotted on top of the image already there. Then, turning the knob changes the value of a parameter, and a different curve results, *but we do not have to replot the grid, axes and labels*. We merely need to `GLOAD` the image (which has everything but the curve and the current parameter value). This allows the curve to be inspected almost in real time. (Run the example program in file `Gstore`.)

The curve looks like the following display:



Using `GSTORE` and `GLOAD` (`Gstore`)

Data-Driven Plotting

Often, data points do not form a continuous line, so one must have the ability to control the pen's position. Earlier, we mentioned a third parameter in the PLOT statement. 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, not just one continuous line.

When using a single X-position and Y-position in a PLOT statement (as opposed to plotting an entire array), 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. The evenness/oddness of the number 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.

Pen Control Parameters

	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. Indeed, this is what happened in the first example in the section “Why Graphics?”. (Zero is considered positive.)

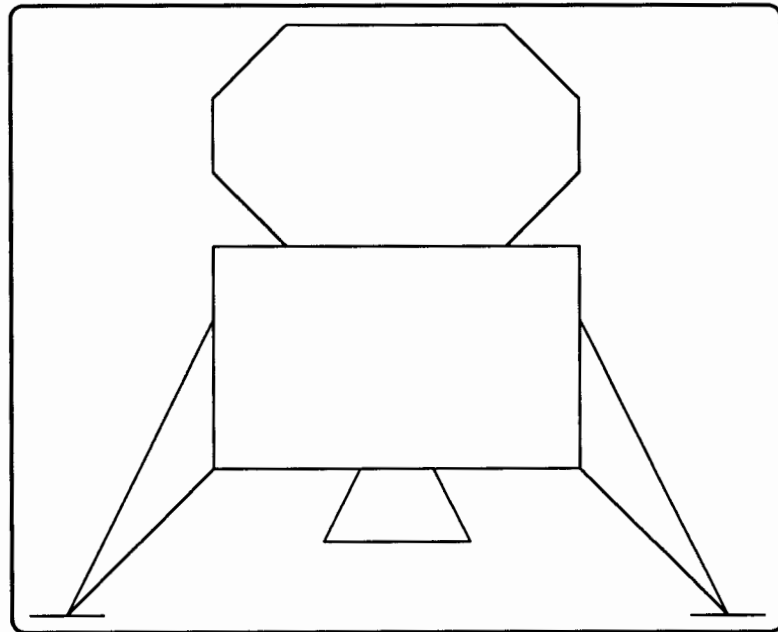
Following is a program which uses pen control. It draws a LEM (Lunar Excursion Module). In particular, see how the PLOT statement was used with an array specifier. Notice that the X and Y values are in the same array as the pen-control parameters. (Run the example program in file Lem1.)

```
100 CLEAR SCREEN                ! Clear the alpha display
110 OPTION BASE 1                ! Arrays start at one
120 DIM Lem(33,3)                ! Data and pen-control array
130 READ Lem(*)                  ! Define the LEM data
140 GINIT                         ! Initialize various graphics parameters
150 PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
160 SHOW -10,10,-10,10          ! Isotropic scaling
170 GRAPHICS ON                  ! Turn on the graphics screen
```

```

180 AREA INTENSITY .125,.125,.125 ! 12.5% gray
190 PLOT Lem(*) ! Plot the data
200 Lem:! X Y Pen | X Y Pen | X Y Pen | X Y Pen
210 DATA 0, 0, 11 ! Start of polygon with FILL and EDGE
220 DATA 1.5, 1, -2, 2.5, 2, -1, 2.5, 3, -1, 1.5, 4, -1 ! Octagon
230 DATA -1.5, 4, -1, -2.5, 3, -1, -2.5, 2, -1, -1.5, 1, -1
240 DATA 0, 0, 7 ! End of first polygon
250 DATA 0, 0, 6 ! Start of polygon with FILL
260 DATA -2.5, 1, -2, 2.5, 1, -1, 2.5, -2, -1, -2.5, -2, -1 ! Box
270 DATA -2.5, 1, -1
280 DATA 0, 0, 7 ! End of second polygon
290 DATA -2.5, -2, -2, -4.5, -4, -1, -2.5, 0, -1, -5, -4, -2 ! Left Leg
300 DATA -4, -4, -1
310 DATA 2.5, -2, -2, 4.5, -4, -1, 2.5, 0, -1, 5, -4, -2 ! Rt. leg
320 DATA 4, -4, -1
330 DATA 0, 0, 10 ! Start of polygon with EDGE
340 DATA -0.5, -2, -2, -1, -3, -1, 1, -3, -1, 0.5, -2, -1 ! Nozzle
350 DATA 0, 0, 7 ! End of third polygon
360 END

```



Example of Data-Driven Plotting (Lem2)

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. If you're pressed for memory, INTEGER numbers take only one-fourth the memory REAL numbers take to store.

PLOT can plot an entire array in one statement, but you must have just one array holding both the data and pen-control parameters. That is, you cannot have the data in a two-column REAL array and the pen-control parameters in a one-column INTEGER array, unless you are plotting one point at a time, as above. The array plotted must be a single two-column or three-column array. If it is a two-column array, the pen-control parameter is assumed to be +1 for every point (pen down after move). If you have a third column in the array, the array columns are interpreted in these ways:

Pen Control when Plotting Entire Arrays

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

8

For a detailed description of these parameters, see IPLOT, PLOT, RPLOT, or SYMBOL in the *HP BASIC Language Reference* manual.

AREA INTENSITY lets you get 17 shades of gray on a black-and-white CRT with an electron gun that is either fully on or completely off. This is done

through a process called **dithering**. Dithering is accomplished by selecting small groups of pixels—a 4×4 square of them on the Series 200/300 computers. Various pixels in the dithering box are turned on and off to arrive at an “average” shade of gray. You can have none of them on, one of them on, two of them on, and so forth, up to all sixteen of them on. It makes no difference *which* pixels are on; they are chosen to minimize the striped or polka-dotted pattern inherent to a dithered image.

For more detail on AREA INTENSITY and other color-related statements, see “Color Graphics.”

Translating and Rotating a Drawing

Often, a segment of a drawing must be replicated in many places. It is possible, but rather tedious to do using the PLOT statement. Another statement called RPLOT, 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 or PDIR statement before the RPLOT. The single parameter for a PIVOT or PDIR 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.

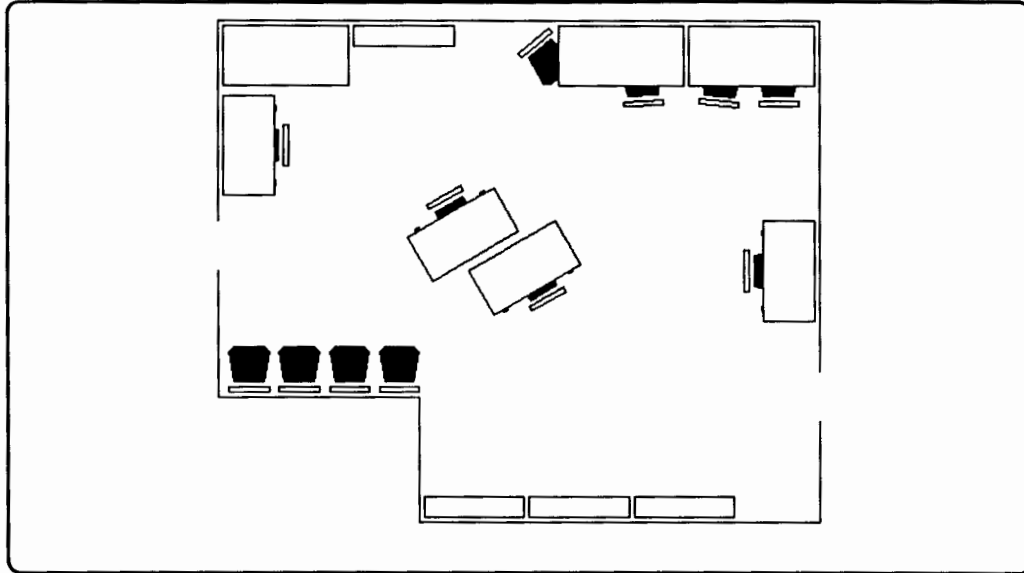
Here is a program using an RPLOT. Various figures are defined with DATA statements: a desk, a chair, a table, and a bookshelf. The program displays a floor layout. Here again, the "end polygon mode" codes (the 0,0,7s in the desk and chair definitions) are unnecessary; when a polygon mode starts, any previous one ends by necessity. (Run the example program in file Rplot.)

```
100 CLEAR SCREEN                ! Clear the alpha display
110 OPTION BASE 1                ! Make arrays start at one
120 DIM Room(10,3),Desk(18,3),Chair(14,3),Bookshelf(4,3),Table(4,3)
130 READ Room(*),Desk(*),Chair(*),Bookshelf(*),Table(*)
140 GINIT                        ! Initialize graphics parameters
150 PLOTTER IS CRT,"INTERNAL"    ! Use the internal screen
160 GRAPHICS ON                  ! Display the graphics screen
170 SHOW 0,120,-10,100          ! Need isotropic units for a map
180 PLOT Room(*)                 ! Draw outline of room
190 DEG                          ! Set degrees mode for angles
200 READ Object$                 ! What to draw?
210 WHILE Object$<>"***STOP***" ! Until done ...
220   READ X,Y,Angle             ! Read where and at what angle
230   MOVE X,Y                   ! Move in unrotated coordinates
240   PIVOT Angle                ! Set rotation for RPLOTS
250   SELECT Object$
260     CASE "Desk"
270       AREA INTENSITY .125,.125,.125 ! 87.5% gray: dark gray
280       RPLOT Desk(*)
290     CASE "Chair"
300       AREA INTENSITY .5,.5,.5 ! 50% gray: half-and-half
310       RPLOT Chair(*)
320     CASE "Bookshelf"
330       RPLOT Bookshelf(*),EDGE
340     CASE "Table"
350       AREA INTENSITY 0,0,0 ! 100% gray scale: Black
360       RPLOT Table(*),FILL,EDGE
370   END SELECT
380   READ Object$
390 END WHILE
400 Room:  DATA 0,60,-2, 0,100,-1, 120,100,-1, 120,30,-1
410        DATA 120,20,-2, 120,0,-1, 40,0,-1, 40,25,-1
420        DATA 0,25,-1, 0,50,-1
430 Desk:  DATA 0,0,11, 0,0,-2, 20,0,-1, 20,-10,-1, 0,-10,-1, 0,0,7
440        DATA 0,0,10, 2,-10,-2, 2,-10.5,-1, 3,-10.5,-1, 3,-10,-1, 0,0,7
```

```

450          DATA 0,0,10, 17,-10,-2, 17,-10.5,-1, 18,-10.5,-1,18,-10,-1, 0,0,7
460 Chair:   DATA 0,0,11, -3,9,-2,  3,9,-1,   4,8,-1,   3,2,-1
470          DATA -3,2,-1, -4,8,-1,  0,0,7
480          DATA 0,0,10, -4,1,-2,  4,1,-1,   4,0,-1,   -4,0,-1,  0,0,7
490 Bookshelf:DATA 0,0,-2, 20,0,-1, 20,-4,-1,  0,-4,-1
500 Table:   DATA 0,0,-2, 25,0,-1, 25,-12,-1,  0,-12,-1
510 Objects: DATA Chair,  14,75,90  ! \
520          DATA Desk,   1,65,90  ! > Upper left corner of room
530          DATA Table,  1,99,0   ! /
540          DATA Bookshelf,27,99,0 !/
550          DATA Chair,   66,44,30 ! \
560          DATA Desk,   50,50,30 ! > Center of the room
570          DATA Chair,  45,65,210 ! /
580          DATA Desk,   60,58,210 !/
590          DATA Bookshelf,41,5,0  !\
600          DATA Bookshelf,62,5,0  ! > Bottom center of room
610          DATA Bookshelf,83,5,0  !/
620          DATA Chair,   6,26,0   !\
630          DATA Chair,  16,26,0   ! \
640          DATA Chair,  26,26,0   ! > Four chairs by west door
650          DATA Chair,  36,26,0   ! /
660          DATA Chair,  63,96,220 ! \
670          DATA Chair,  85,83,3   ! > Four chairs by northeast tables
680          DATA Chair,  112,83,0  ! /
690          DATA Chair,  100,83,355 !/
700          DATA Table,  68,99,0   ! \
710          DATA Table,  94,99,0   ! > Two tables in upper right
720          DATA Chair,  105,50,270 ! \
730          DATA Desk,   119,60,270 ! > Desk and chair by east door
740          DATA ***STOP***
750          END

```



Relative Plotting of a Floor Layout (Rplot)

Notice that you can specify the `EDGE` and/or `FILL` parameters in the `RPLOT` statement itself, in addition to in the array. (`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 both in the `PLOT` statement and in the data, and the instructions differ, the value in the data replaces `FILL` or `EDGE` on the statement.

Also notice 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.

8

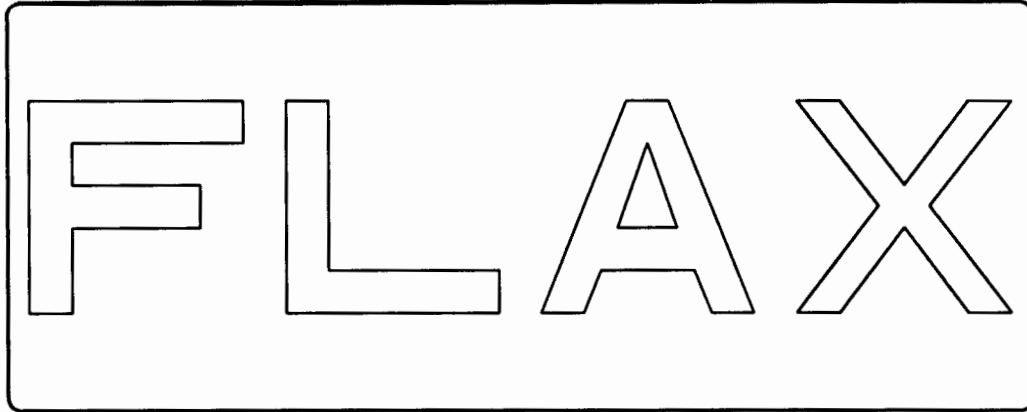
Incremental Plotting

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.

Below is an example program using IPLOTs. It reads data from statements describing the outlines of certain letters of the alphabet, and then plots them. (Run the example program in file Iplot.)

```
100 CLEAR SCREEN           ! Clear the alpha display
110 OPTION BASE 1         ! Make the arrays start at 1
120 DIM Array(20,3)       ! Set aside space for the array
130 GINIT                 ! Initialize various graphics parameters
140 PLOTTER IS CRT,"INTERNAL" ! Use the internal screen
150 GRAPHICS ON           ! Turn on graphics screen
160 SHOW 1,35,-15,15     ! Isotropic scaling
170 FOR Letter=1 TO 4     ! Four letters total
180   READ Points         ! How many points in this letter?
190   REDIM Array(Points,3) ! Adjust the array size accordingly
200   READ Array(*)       ! Read the correct number of points
210   MOVE Letter*6,0     ! Move to lower-left corner of letter
220   IPLOT Array(*)     ! Draw letter
230 NEXT Letter          ! et cetera
240 F: DATA 10, 0,5,-1, 5,0,-1, 0,-1,-1, -4,0,-1, 0,-1,-1
250 DATA 3,0,-1, 0,-1,-1, -3,0,-1, 0,-2,-1, -1,0,-1
260 L: DATA 6, 0,5,-1, 1,0,-1, 0,-4,-1, 4,0,-1, 0,-1,-1
270 DATA -5,0,-1
280 A: DATA 12, 2,5,-1, 1,0,-1, 2,-5,-1, -1,0,-1, -.4,1,-1
290 DATA -2.2,0,-1, -.4,-1,-1, -1,0,-1, 1.8,2,-2, .7,2,-1
300 DATA .7,-2,-1, -1.4,0,-1
310 X: DATA 12, 1.9,2.5,-1, -1.9,2.5,-1, 1,0,-1, 1.5,-2,-1, 1.5,2,-1
320 DATA 1,0,-1, -1.9,-2.5,-1,1.9,-2.5,-1, -1,0,-1, -1.5,2,-1
330 DATA -1.5,-2,-1, -1,0,-1
340 END
```



Incrementally Plotting Letters (lplot)

Drawing Polygons

When you want a regular polygon, or a part of one, drawn on the screen, two statements will help. The first is called POLYGON. (In this discussion, polygons drawn when anisotropic units are in effect are also considered “regular.”)

One attribute of POLYGON is that *it forces polygon closure*; that is, the first vertex is connected to the last vertex, so there is always an inside and an outside area. This is true even when drawing only one side of a polygon, in which case a “single” line results. This is actually two lines, from the first point to the last point, and back to the first.

There are two final keywords which may be included in a POLYGON statement: FILL causes the interior of the polygon or polygon segment to be filled with the current fill color as defined by AREA PEN, AREA COLOR, or AREA INTENSITY. FILL specified without EDGE causes the interior of the polygon to be indistinguishable from the edge. EDGE causes the edges of the polygon to be drawn using the current pen and line type. If both FILL and EDGE are specified (and FILL must be first), the interior will be filled, then the edge will be drawn. If neither FILL nor EDGE is specified, EDGE is assumed. On an HPGL plotter, only EDGE works.

Polygons can be rotated by specifying a non-zero value in a PIVOT or PDIR statement before the POLYGON statement is executed. Also, a PDIR

statement can be used to specify the angle of rotation. PDIR works with IPLOT, RPLOT, POLYLINE, POLYGON, and RECTANGLE. The rotation occurs about the origin of the figure. For example, PDIR 15 would rotate a figure 15 units (degrees, radians).

The shape of the polygon is affected by the viewing transformation specified by SHOW or WINDOW. Therefore, anisotropic scaling causes the polygon to be stretched or compressed along the X and Y axes.

Pen status also affects the way a POLYGON statement works. If the pen is up at the time POLYGON is specified, the first vertex specified is connected to the last vertex specified, *not* including the center of the polygon, which is the current pen position. If the pen is down, however, the center of the polygon is also included in it. Thus, for piece-of-pie shaped polygon segments, like those in pie charts, cause the pen to be down before the POLYGON statement is executed. After POLYGON has executed, the current pen position is in the same position it was before the statement was executed, and the pen is up.

But I Don't Want Polygon Closure ...

Another statement, POLYLINE, acts much the same way as POLYGON, except it does not connect the last vertex to the first vertex; it does not *close* the polygon. Since there is no "inside" or "outside," it is meaningless to use FILL or EDGE.

As in the case of POLYGON, a PIVOT or PDIR statement prior to executing POLYLINE will rotate the figure. Anisotropic scaling will stretch or compress the figure along the axes, and if the pen is down prior to invoking the statement, a line will be drawn from the center to the first perimeter point. After POLYLINE has executed, the current pen position is the same as it was before the statement was executed, and the pen is up.

Rectangles

One of the most used polygons in computer graphics is the rectangle. You can draw a rectangle by moving to the point where you want one of the corners to be and then specifying which directions to proceed to from there, first in the X direction, then in the Y direction. Which corner of the rectangle ends up at the current pen position depends on the signs of the X and Y parameters. For example, if you want a rectangle with a lower left corner at 3,2, which is 4 units wide and 5 units high, there are four ways you could go about it:

```
MOVE 3,2      (Reference point is the lower left corner)  
RECTANGLE 4,5
```

OR

```
MOVE 7,2      (Reference point is the lower right corner)  
RECTANGLE -4,5
```

OR

```
MOVE 3,7      (Reference point is the upper left corner)  
RECTANGLE 4,-5
```

OR

```
MOVE 7,7      (Reference point is the upper right corner)  
RECTANGLE -4,-5
```

Again, you can specify FILL, EDGE, or both. FILL will fill the rectangle with the current color as specified by AREA PEN, AREA COLOR, or AREA INTENSITY. EDGE causes the edge of the rectangle to be drawn in the current pen color and line type. If both are specified, FILL must be specified first. If neither is specified, EDGE is assumed. The current pen position is not changed by this statement, and pen status prior to execution makes no difference in the resulting rectangle.

User-Defined Characters

For many special-purpose programs, there is a shortage of characters that can be displayed on the screen. Greek letters— Δ , π , Σ , and so forth—are often needed for mathematic communication, as are non-alphabetic symbols like $\sqrt{\quad}$, ∞ , and \pm . To alleviate this shortage of symbols, the SYMBOL statement allows you to draw any definable character. In function, it is similar to PLOT using an array, except the figure drawn by SYMBOL is subject to the three transformations which deal with character labeling: CSIZE, LDIR, and LORG.

8

Note



User-defined alpha character fonts are available on bit-mapped alpha displays. See “Communicating with the Operator” in *HP BASIC 6.2 Advanced Programming Techniques*.

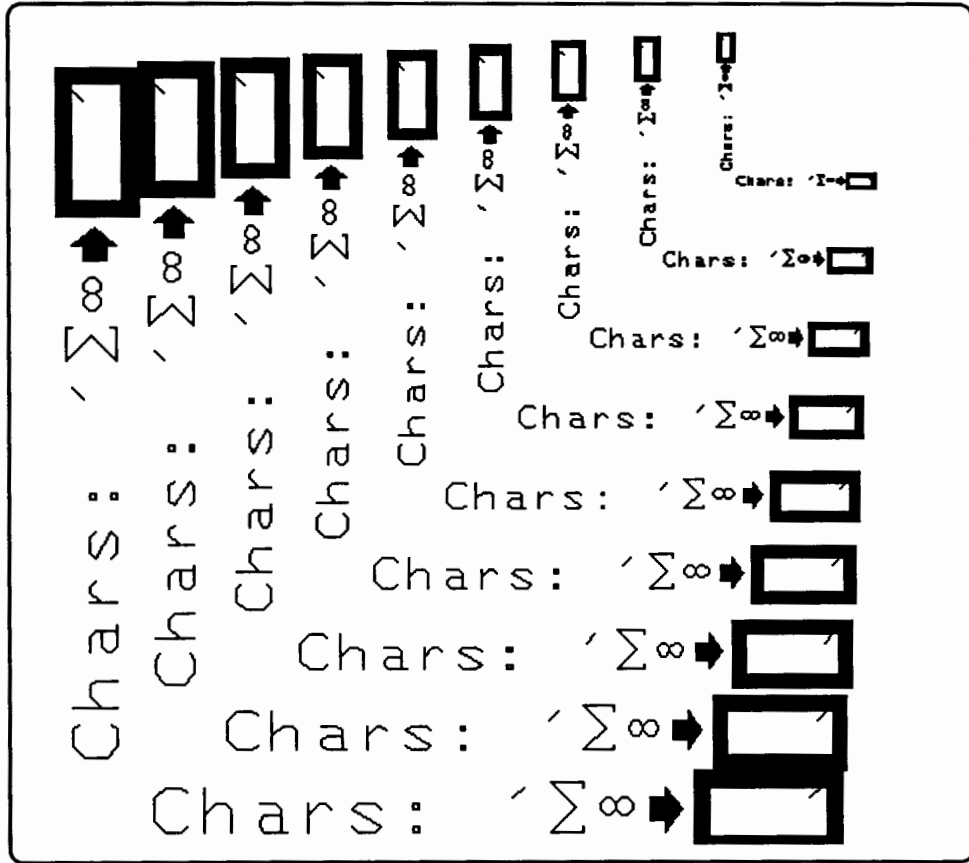
The first argument needed by the SYMBOL statement is the array containing the instructions on what to draw. As in PLOT, this array may have two or

three columns. If the third column does not exist, it is assumed to be +1 for every row of the array. If it does exist, the valid values for the third-column entries are identical to those for PLOT, RPLOT, and IPLOT when using an array. The possible values for the third column are listed earlier in this section in the table titled "Pen Control for Plotting Entire Arrays." For more detail on the meaning of these values, see the *HP BASIC Language Reference* manual.

Moves and draws specified in an array for a SYMBOL statement are defined in the **symbol coordinate system**. This coordinate system is a character cell, a 9×15 rectangle. Figures drawn in this coordinate system may be filled, edged, or both. FILL and EDGE can appear in the SYMBOL statement itself, or they can be specified in the data array. If FILL and/or EDGE are specified in both places, the instruction in the data array overrides that of the statement.

One interesting feature of this statement is that values *outside* the character cell boundaries are valid. Thus, you can define characters that are several lines high, several characters wide, or both. This feature is used in the following program. The SYMBOL statement, by virtue of its syntax, can only be used for one user-defined character at a time; the pen must be moved to the new position before each character. Therefore, UDCs cannot be embedded in a string of text. If the situation remained this way, the utility of the SYMBOL statement would be limited by its cumbersome implementation.

The program `Symbol` makes UDCs easier to use. `Symbol` is a special-purpose program which calls two general-purpose subprograms. The first subprogram (`New_udc`) is called to define a new UDC. Its parameters are: 1) the character to be replaced by the UDC, and 2) the array defining the character. The second subprogram (`Label`) is called after all desired UDCs have been defined. This allows text to be labeled (written in graphics mode), intermixing ASCII characters with user-defined characters at will. As mentioned above, all user-defined characters are affected by `CSIZE`, `LDIR` and `LORG`, so no matter how the label is written, the UDCs will act properly. (Run the example program in file `Symbol`.)



Implementing User-Defined Characters (Symbol)

8

Of course, the limits (20 UDCs and 30 rows maximum) may be reduced or expanded to fit your purpose. Note that in lines 180 through 210 of the program, characters 168 through 171 were replaced by the four UDCs. There is nothing magical about these four characters. The characters replaced could have been any characters between 0 and 255, and they need not be consecutive.

Also note that in line 450 of the program, there are two spaces after the `CHR$(171)`. This is because character 171 was replaced by the box, which is three character cells wide. The two extra spaces prevent the right two-thirds of the box from being overwritten by whatever is labeled after it.

Multi-Plane Bit-Mapped Displays

When using multi-plane (color or gray-scale) bit-mapped displays, BASIC provides the ability to specify which planes to write-enable for alpha and graphics and which planes to display. This feature provides several useful features on bit-mapped displays. Before we look at the uses, however, let's look at what these masks are:

- The graphics write-enable mask
- The display-enable mask
- The alpha write- and display-enable masks (similar in structure and operation to the graphics write- and display-enable masks, so little is said here about it)
- Interactions between the alpha masks and graphics masks

The Graphics Write-Enable Mask

First, we'll look at the **graphics write-enable mask**. Its function is to specify which frame buffer planes are to be written to by graphics operations.

To illustrate, assume that your machine has four planes in the frame buffer. Suppose you want to set the graphics write-enable mask such that graphics operations use only the first two planes of the frame buffer. This is done by setting the first element of an integer array to the value desired, and then executing the **GESCAPE** with operation selector 7:

```
INTEGER Graphics_masks(0:0)
Graphics_masks(0)=IVAL("0011",2)      ! Set graphics write mask: planes 1&2
GESCAPE CRT,7,Graphics_masks(*)      ! Set write mask; display mask is
unchanged
```

The graphics write-enable mask has the value 0011 (in decimal, the value is 3, but the masks will be shown in binary for clarity). This indicates that only planes 1 and 2 of the frame buffer will be used for graphics write operations. For example, drawing a line can change bits only in planes 1 and 2. A graphics write-enable mask of 0011 also implies that there are only four color-map "pens" available for use by graphics operations: 0 through 3.

The Graphics Display-Enable Mask

The difference between the graphics write-enable mask and the graphics display-enable mask is that the former specifies which planes are actually modified by graphics operations (regardless of whether or not they are displayed), and the latter specifies which planes can be seen by the user (regardless of whether or not anything has been or can be written to them).

Suppose you want to set the graphics display-enable mask such that only the contents of the first two planes of the frame buffer are visible to the user. This is done by setting the *second* element of an integer array to the value desired, and then executing GESCAPE with operation selector 7:

```
INTEGER Graphics_masks(0:1)
Graphics_masks(0)=<some value>
Graphics_masks(1)=IVAL("0011",2)    ! Set display-enable mask: planes 1&2
GESCAPE CRT,7,Graphics_masks(*)    ! Set masks
```

Although for many operations the graphics write-enable mask and the graphics display-enable mask will have the same value, they need not be the same. There are many instances where they will be different. In many of these cases, one or both of the graphics masks will change regularly as the program executes.

The Alpha Masks

The alpha write-enable mask and display-enable mask do the same jobs for the alpha display as their graphics counterparts do for the graphics display.

Note



When the PLOTTER IS device *is the same as* the ALPHA CRT device, the alpha display mask *is* the graphics display mask and vice versa. Though they are the same entity, the mask can be accessed either by GESCAPE CRT,7 or CONTROL CRT,20 (see below).

8

To set and read the alpha write-enable mask, use the following statements:

```
SET ALPHA MASK IVAL("1100",2)    ! Set alpha WRITE-enable mask: planes 3&4
STATUS CRT,18;Alpha_writemask    ! Read alpha WRITE-enable mask
```

To set and read the alpha display-enable mask, use the following statements:

```
SET DISPLAY MASK IVAL("1100",2) ! Set alpha DISPLAY-enable mask: planes 3&4
STATUS CRT,20;Alpha_disp_mask ! Read alpha DISPLAY-enable mask
```

Interactions Between Alpha and Graphics Masks

The alpha and graphics write-enable masks both default at power-up and SCRATCH A to all planes in the machine. Thus (assuming you have a four-plane machine), both write-enable masks would be 1111. All four planes are at once *both* alpha planes and graphics planes. This implies that when you write graphical information, the alpha display may be modified and overwritten, and vice versa.

Alternately, if you set the alpha write-enable mask to 0011 (alpha uses planes 1 and 2), and the graphics display-enable mask to 1000 (graphics uses plane 4), plane 3 is neither alpha nor graphics.

This is a departure from older machine architectures where there was a fixed alpha area and a fixed graphics area, and in which "alpha" equals "not graphics" and "graphics" equals "not alpha." Now, planes can be "both alpha and graphics" or "neither alpha nor graphics," and you can choose where to put them.

When a plane is designated as both an alpha plane and a graphics plane (default state), alpha and graphics share the bit-mapped plane so that writing to the display overwrites existing information. A plane designated as graphics only can be written to by graphics statements such as DRAW and LABEL, but not with alpha statements. Similarly, an alpha-only plane can be written to with statements such as PRINT and DISP, but not with graphics statements. Note that when executing PLOTTER IS, PENs with bits in non-graphics planes are not updated when the color map is initialized. The graphics write-enable mask also affects GSTORE and GLOAD; see the *HP BASIC Language Reference* for details.

With this feature you can simulate separate alpha and graphics planes for compatibility with older systems that do not have bit-mapped alpha. Assuming you have a four-plane display, you could designate planes 1 through 3 for graphics, and plane 4 for alpha. This gives you eight pure graphics colors, and a single alpha color, and some of the capabilities of a separate alpha/graphics system on bit-mapped hardware:

- Turning alpha and graphics on and off independently.
- Dumping the graphics display or the alpha display independently.
- Scrolling alpha without scrolling graphics along with it.

It has always been true that you could dump alpha when using bit-mapped displays. However, this capability is afforded by the presence of an “alpha buffer,” a spare storage place for all alpha information. This enables alpha to be dumped to a printer that does not have raster graphics capabilities.

There is one tricky condition that may occur if you’re not careful. Suppose your program has executed ALPHA PEN 1, and at some point it changes the alpha write-enable mask to 1100. Suddenly, the output stops appearing, the run light remains on, and the “live” keyboard appears dead. By all appearances, your machine is hung. This is not the case, however.

What actually happens is this. Before the write-enable mask was changed, everything was going as expected. When you changed the write-enable mask to 1100, you caused the machine to only turn on pixels when using pens whose numbers had some bits in common with the write-enable mask, now 1100. The only pens which satisfy this condition are pens 4 through 15. In other words, nothing appeared when you were “writing” because on one hand you said “write pixels in planes 3 and 4 only,” and on the other hand you said “I am going to write with a pen which doesn’t turn on pixels in planes 3 and 4.”

The following program illustrates the concept of simulating separate alpha and graphics. (You can also use SEPARATE ALPHA FROM GRAPHICS and MERGE ALPHA WITH GRAPHICS to accomplish these tasks; however, this program shows the intermediate steps in how the alpha write-enable and display-enable masks interact). The program writes some text which will be visible. Next, it changes the display mask so the alpha display mask and the current pen are disjoint. After writing a bit more, the program pauses, so you can see just how hung it really looks. When you press **CONTINUE** (**f2**) in the

System menu of an ITF keyboard), the alpha display mask and the pen will once again come into agreement.

```
10  ! This program demonstrates proper and improper use of alpha masks.
20  ! (Return everything to its default state if the program didn't finish)
30  SET ALPHA MASK IVAL("1111",2)      ! Set alpha write-enable mask.
40  SET DISPLAY MASK IVAL("1111",2)    ! Set alpha display-enable mask.
50  CLEAR SCREEN
60  GINIT
70  PLOTTER IS CRT,"INTERNAL";COLOR MAP
80  ! (Start the demo)
90  ALPHA PEN IVAL("0011",2)          ! Set alpha pen
100 SET ALPHA MASK IVAL("0011",2)     ! Set alpha write-enable mask.
110 SET DISPLAY MASK IVAL("0011",2)   ! Set alpha display-enable mask.
120 PRINT "I'm printing visibly."
130 WAIT 4
140 PRINT "Changing the alpha display mask -- this text will vanish."
150 WAIT 4
160 SET DISPLAY MASK IVAL("1100",2)   ! Set alpha display-enable mask.
170 WAIT 4
180 FOR I=1 TO 10
190   PRINT " I'm printing stuff now but you can't see it."
200 NEXT I
210 SET DISPLAY MASK IVAL("0011",2)   ! Set alpha display-enable mask.
220 WAIT 1
230 PRINT "The above text just became visible."
240 WAIT 4
250 PRINT "The above text will disappear again and the machine will"
260 PRINT "appear hung until you press CONTINUE."
270 PRINT "(Note the run light stays on after the computer pauses.)"
280 WAIT 9
290 SET DISPLAY MASK IVAL("1100",2)   ! Set alpha display-enable mask.
300 PAUSE
310 SET DISPLAY MASK IVAL("0011",2)   ! Set alpha display-enable mask.
320 WAIT 1
330 PRINT
340 PRINT "The above text just became visible again."
350 WAIT 4
360 SET ALPHA MASK IVAL("1111",2)     ! Set alpha write-enable mask.
370 SET DISPLAY MASK IVAL("1111",2)   ! Set alpha display-enable mask.
380 ALPHA PEN IVAL("0001",2)         ! Set alpha pen
390 PRINT "I just set the alpha masks and pen back to normal."
400 END
```



Another use of the graphics write-enable mask is the fast display of multiple pictures. For example, you could:

1. Disable all planes displayed. This makes the screen blank.
2. Enable plane 1 for writing.
3. Create a single-color picture in plane 1.
4. Enable plane 1 for display. Now the picture in plane 1 appears on the screen.
5. Disable plane 1 for writing and enable plane 2.
6. Create a different single-color picture in plane 2.
7. Disable plane 1 for display and enable plane 2. Now the picture in plane 2 appears on the screen.

Cycling through this sort of a loop—flashing consecutive pictures on the screen—simulates a rudimentary animation; i.e., “motion” pictures. Be aware, however, that the drawing speed is much too slow to describe smooth, non-jerky movements.

The selection of graphics planes to be write-enabled and display-enabled is accessed via the GESCAPE statement (in the GRAPHX binary).

Disabling and Enabling Alpha Scrolling

Series 300 computers (and Series 200 Model 237 computer) have alpha and graphics on the same raster (this is called “bit-mapped alpha”). One of the most important implications of this architecture for graphics is that when you scroll alpha information, you also scroll graphics. If you want to disable scrolling of both alpha and graphics due to the cursor-control keys (such as  and ) , you can execute this statement:

```
CONTROL KBD,16;1
```

If scrolling is currently disabled and you want to re-enable it, execute this statement:

```
CONTROL KBD,16;0
```

On non-bit-mapped alpha displays, **CONTROL KBD 16, 1** will also disable alpha scrolling. Because the alpha and graphics are physically separate, however, graphics will not be affected by alpha scrolling on non-bit-mapped displays.

Introduction to Color Graphics

Color can be used for emphasis, clarity, and to present visually pleasing images. This section is an overview of color graphics on the computer. For a detailed discussion of color graphics programming techniques, refer to the “Color and Gray Scale Graphics” chapter in *HP BASIC 6.2 Advanced Programming Techniques*.

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 Dominant-Pen Mode

Pen	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.

Color Mapped Color

8

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 Values

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 were chosen by a graphics designer to avoid clashing with each other. They are intended for business charts and graphs, and similar applications.

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.

For a much more detailed discussion of this topic, refer to the *HP BASIC 6.2 Advanced Programming Techniques* manual.

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.



Using Printers and Plotters for Generating Graphics

The preceding chapter described how to generate graphic images on your CRT display. In this chapter, we discuss selecting external printing and plotting devices which you can use to generate graphics on paper. (See the *HP BASIC 6.2 Advanced Programming Techniques* manual for more information on “external” CRT displays that can be connected to your computer through display interfaces.)

- First we’ll show how to *dump graphics* images from a CRT display to a printer.
- Then we’ll show how to use the PLOTTER IS statement to select HP-IB plotters, which may be connected through the built-in HP-IB (Hewlett-Packard Interface Bus) port in the back of your computer. This section also describes how to use “Hewlett-Packard Graphics Language” (HPGL) commands to talk directly to plotters.
- The next section shows how to send plotting commands to a file.
- The last part of the chapter describes using SRM plotter spoolers.

Note



The *Manual Examples* disk (`/usr/lib/rmb/demo/manex` for BASIC/UX users) contains programs found in this chapter. As you read through the following sections, load the appropriate program and run it. Experiment with the programs until you are familiar with the demonstrated concepts and techniques. Remember that on BASIC/WS, some of the programs require the MAT (matrix) BIN file.

Dumping Raster Images to Printers

After generating an image on your CRT display, you can print it on paper using a printer. This operation is called a **graphics dump** or **screen dump**. It is accomplished by copying data from the frame buffer to be reproduced dot for dot on the printer. For information on putting a screen dump into a file, read the section "Dumping Raster Images to a File."

First, the image must be generated on any CRT display. BASIC "takes a snapshot" of the graphics screen at some point in time, and sends it to a printer. BASIC doesn't care *how* the dots were turned on or off. Thus, all CRT graphics capabilities (except color) are available.

Note



You can send a color dump display to a PaintJet™ using the GDUMP.C utility. For more information on this, read the "BASIC Utilities Library" section of the *Installing and Maintaining HP BASIC* manual.

Dumping to HP Raster Interface Standard Devices

If your printer conforms to the HP Raster Interface Standard, dumping graphic images is trivial. For example:

```
100  DUMP DEVICE IS 701
110  DUMP GRAPHICS
```

or

```
100  DUMP GRAPHICS #701
```

To determine whether or not your printer conforms to this standard, see your printer's manual or the configuration reference.

Both of these program segments would take the image last specified in the graphics frame buffer (the internal CRT display by default) and send it to the printer at address 701. (If no source device is specified, the image is taken from the last active CRT or it is the default graphics display.) Address 701 is the default factory setting for printers.

Dumping from a Color Display

When dumping an image from a color display to a printer, the state of each bit sent to the dump device is calculated by performing an inclusive OR operation on all color-plane bits for each pixel (in other words, a dot is printed if the pen used to write the dot was not equal to 0). Thus, no color information is dumped.

Dumping from a High-Resolution Display

If the source device is a high-resolution display, the image will not fit on one page of most HP printers. In such cases, use only the portion of the screen that is dumped to the printer. You can see which portion is not used by drawing a grid, for instance, and then dumping it to the printer. When you see which part is not displayed on the printer, you can change the region of the graphics display (with VIEWPORT, for example) so that you do not use that region.

Using the DUMP GRAPHICS Key

The **DUMP GRAPHICS** key (on an ITF keyboard, press **Shift** and the unlabeled key above the + on the numeric keypad) will also dump 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.

Aborting Graphics Dumps

If a DUMP GRAPHICS operation is aborted with the **CLR I/O** key (**Break** on an ITF keyboard), the printer may or may not terminate its graphics mode. Sending 192 null characters (ASCII code zero) to a printer such as an HP 9876 terminates its graphics mode. For example:

```
OUTPUT Dump_dev USING "#,K";RPT$(CHR$(0),192)
```

To dump a graphics image from an external color monitor that is interfaced through an HP 98627A at address 28, you could execute either of the following:

```
DUMP DEVICE IS Dump_dev  
DUMP GRAPHICS 28
```

or

```
DUMP GRAPHICS 28 TO #Dump_dev
```


Expanded Dumps

If you want the image to be twice as large in each dimension as the actual screen size, you can execute the following two statements:

```
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 2x2 square of dots, and the resulting image is rotated 90° to allow more of it to fit on the page. Depending on your screen size and printer size, the image being dumped may not entirely fit on the printer. If it doesn't, it will be truncated.

Dumping Displays with Non-Square Pixels

If you have an HP 98542A or HP 98543A display, the pixels are not square. Images are not distorted because of this, however, because the BASIC system compensates for the rectangularity. Also, the image is not distorted when dumping graphics because the BASIC system compensates for the non-square pixels.

For machines which have a display with non-square pixels, a non-expanded DUMP GRAPHICS will produce an image that matches the CRT *only* if no alpha appears in the graphics planes. Since most printers print square pixels, this routine treats graphics pixel pairs as single elements and prints one square for each pixel pair in the frame buffer. Because alpha works with individual pixels and not with pixel pairs, mixed alpha and graphics will appear blurred on a DUMP GRAPHICS non-expanded output. Using the EXPANDED option causes the vertical length (the height on the CRT) to be doubled as before, but dumps each separate pixel. In this mode, mixed alpha and graphics will appear the same on the dump as on the CRT. Note that on multiplane bit-mapped displays, only graphics write-enabled planes are dumped.

Dumping to Non-Standard Printers

If your printer that does not conform to the HP Raster Interface Standard, all is not lost. It must, however, be capable of printing raster-image bit patterns. There are two main methods by which printers print bit-sequences:

- When a printer receives a series of bits, it prints them in a one-pixel-high line across the screen. The paper then advances one pixel's distance, and the next line is printed.
- The other method (which lends itself to user-defined *characters* more than graphics image dumping) takes a series of bits, breaks it up into 8-bit chunks, and prints them as vertical bars 8 pixels high and 1 pixel wide. The next 8 bits compose the next 1×8-pixel bar, and so forth.

Example of Dumping to an Non-standard Printer

The HP 82905A printer uses the latter method listed above. The image (which is printed sideways) takes a GSTOREd image and breaks the 16-bit integers into two 8-bit bytes, and sends them to the printer one row at a time. This is the reason for the Hi\$ and Lo\$—the high-order (left) and low-order (right) bytes of the current integer. For an example of a subprogram that dumps graphics from a medium-resolution display to an HP82905A printer, see the file `DumpGraph` on the *Manual Examples* disk or in `/usr/lib/rmb/demo/manex`.

To DUMP GRAPHICS to other types of printers, you can modify the preceding subprogram appropriately for the destination device. See your printer's manual for information about its "raster-image" mode.

Negative Images

Note that on a CRT, an “on” pixel is light on an otherwise dark background, and on a printer, an “on” pixel is dark on an otherwise light background. Thus, the hard copy is a negative image of the image on the screen. To dump light images on a dark background, you can use the BINCOMP function to complement the bits in every word before you send the image to the printer, or you can invert the bits of an integer by using this program segment:

```
IF N=-32768 THEN
  N=32767
ELSE
  N=-N-1
END IF
```

The reason for the subtraction is that Series 200/300 computers use two’s-complement representation of integers. Also, -32 768 is a special case because you cannot negate -32 768 in an integer; +32 768 cannot be represented in a signed, sixteen-bit, two’s-complement number.

Dumping Raster Images to a File

This section covers the keywords required to dump an alphanumeric or graphics display to a specified file.

Specifying the File that Receives the Raster Dump

The DUMP DEVICE IS statement specifies which file receives the data when either a DUMP ALPHA or DUMP GRAPHICS statement is executed without a device selector. The syntax used for DUMP DEVICE IS is as follows:

```
DUMP DEVICE IS file_specifier, [EXPANDED]
```

where the *file_specifier* must be a BDAT or an HP-UX file which is to receive the output from the raster dump. The optional parameter EXPANDED doubles the size of the image and rotates it 90 degrees.

Example

This example dumps a raster image to a BDAT file and then dumps that file to a printer.

```
100 ! Draw a picture and dump it to a BDAT file named "Picture"
110 ! then output the file to a printer.
120 INTEGER I
130 CREATE BDAT "Picture",1 ! Creates a file named "Picture".
140 DUMP DEVICE IS "Picture" ! Assigns dump device to be the file
150 ! named "Picture".
160 ! Put statements for drawing your picture here.
.
.
.
300 DUMP GRAPHICS ! Dump raster images to the file
310 ! named "Picture".
320 ! Assign file names for dumping the file "Picture" to a
330 ! printer.
340 ASSIGN @File TO "Picture";FORMAT OFF
350 ASSIGN @Printer TO 701;FORMAT OFF
360 ON END @File GOTO Done
370 ! Dump the file "Picture" to the printer.
380 LOOP
390 ENTER @File;I
400 OUTPUT @Printer;I
410 END LOOP
420 Done:!
430 ! Close I/O path names
440 ASSIGN @File TO *
450 ASSIGN @Printer TO *
```

Using Plotters

In Chapter 1, the program listings contained a line which said:

```
PLOTTER IS CRT,"INTERNAL"
```

This caused the computer to activate the internal CRT graphics raster as the plotting device, directing all subsequent commands to the screen.

Selecting a Plotter

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"
```

Specifying HPGL tells BASIC to generate HPGL commands when it executes graphics statements, and to send them to the current plotting device.

Plotter Graphics with HPGL Commands

When you execute BASIC graphics statements on an HP plotter, you can intersperse 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.

```
PLOTTER IS 705,"HPGL"  
FRAME  
GSEND "HPGL command(s)"  
MOVE X,Y  
GSEND "HPGL command(s)"  
DRAW X+10,Y-20
```

HPGL command sequences are terminated by a line-feed, a semicolon, or an EOI indication, which is sent by the HP-IB END keyword. Individual commands within a sequence are typically delimited by semicolons. Note that the GSEND command sends a carriage return/line feed after the specified string.

Many HPGL commands are available, but which ones you will be able to use depend on the device itself. Plotters are not the only devices which use HPGL; some 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 or useful HPGL commands.

Example: Controlling Pen Speed

To control pen velocity, you could have a statement:

```
GSEND "VS10;"
```

9

9-8 Using Printers and Plotters for Generating Graphics

“VS” stands for “Velocity Select” and the “10” specifies centimeters per second. This statement specifies a *maximum* speed rather than an *only* speed. The range and resolution of pen speeds, and default maximum speed depend on the plotter.

Example: 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 to match pen type to drawing medium. An example statement is:

```
GSEND "FS3,6;"
```

This statement (*Force Select*) specifies that pen number 6 should be pressed onto the drawing medium with force number 3. The range in force depends on the plotter.

Example: Selecting Character Sets

Some plotters contain internal character sets which may be more pleasing to the eye or more appropriate for your application than the BASIC character set. For example, to use Character Set 1, type:

```
GSEND "CS1;"
```

Error Detection when Using HPGL Commands

If you make an error when using HPGL commands, 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 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 used.

This is not an exhaustive list of HPGL commands. A thorough understanding of HPGL can only be obtained by combining information from the owner's manual of the particular device you have with actual hands-on experience.

Plotting to Files

The preceding PLOTTER IS statements were used to direct HPGL commands to a plotter. You can also direct these commands to a file. This is useful when you want to check what is being sent to a plotter, or when you want to generate a special sequence of HPGL commands that you cannot generate with BASIC graphics statements. It is also useful when using an SRM plotter spooler, as discussed in a subsequent section.

The following statements would send subsequent plotter output (HPGL commands) a file named Plot.

```
CREATE BDAT "Plot:,700",20
PLOTTER IS "Plot:,700";"HPGL"
FRAME
MOVE 0,0
DRAW 100,100
MOVE 0,100
DRAW 100,0
MOVE 50,50
CSIZE 15
LABEL "Big X"
PLOTTER IS CRT,"INTERNAL"
```

Plot must be a BDAT file. Another PLOTTER IS statement, SCRATCH A, or GINIT statement closes the file. A Reset also closes the file.

Plotter Paper Sizes

The PLOTTER IS statement also lets you specify a non-default paper size. Here is the general syntax you can use; just substitute the limits (in millimeters) for the four parameters:

```
PLOTTER IS "File","HPGL", Xmin, Xmax, Ymin, Ymax
```

See the *HP BASIC Language Reference* and your plotter's manual for additional information.

Limitations

If you perform an operation on one plotting device and send the plot to another device which does not support that operation, it won't work. For example, area fills, which are valid operations on most displays, are not available on plotters. Color map operations, which are valid on displays, are not valid on a plotter. Erasing lines can be done on displays, but, naturally, not on a hard-copy plotter. On the other hand, HPGL commands will be interpreted correctly by a hard-copy plotter, but not by a display.

Using GSEND with PLOTTER IS Files

This statement sends a string of characters to the current PLOTTER IS device, which may be a file or a plotter.

```
GSEND "LBThis is an example HPGL command string."&CHR$(3)&";"
```

The string is to contain HPGL commands. GSEND is useful when the PLOTTER IS device is a file, since it is not possible to OUTPUT an HPGL command to the file while it is the PLOTTER IS device.

```
CREATE BDAT "Plot:,700",20
PLOTTER IS "Plot:,700";"HPGL"
FRAME
MOVE 0,0
DRAW 100,100
MOVE 0,100
DRAW 100,0
MOVE 20,20
GSEND "CS2;"
GSEND "LBThis is X in the plotter's character set 2."&CHR$(3)&";"
PLOTTER IS CRT,"INTERNAL"
```

Note that HPGL syntax is *not* checked by the BASIC system. Therefore, you will need to take extra care when using HPGL commands in this manner.

Using SRM Plotter Spoolers

The SRM system not only provides shared access to plotters, but also manages their use so that workstations never need to wait for output to be generated.

What Are Spoolers?

To use shared plotters, you place files to be output into a special directory where they are held until the plotter is free. This method is called “spooling,” and the directory where the files are kept is called the “spooler directory.” After a file is placed in a spooler directory, the workstation is free to do other processing.

Setting Up a Plotter Spooler

Spooler directories are created for the SRM server’s use when the shared peripherals are installed on the SRM system. Setting up a spooler directory is explained in the “Interfaces and Peripherals” chapter of the *SRM System Manager’s Guide*. The examples in this section assume that a spooler directory named PL (for “PLOTter”) has been created in the SRM root directory.

Preparing Plotters

If your plotter does not feed paper automatically, a message appears on the SRM server’s screen indicating that the plotter needs to be set up. After you put paper in the plotter, you may begin plotting by using the server’s SPOOL CONTINUE command (described in the *SRM System Manager’s Guide*). Plotters with automatic paper feed require no operator intervention.

Plotter Spooling

These are the steps in using the SRM plotter spooler:

1. Create a file.
2. Specify it as the PLOTTER IS device.
3. Perform BASIC or HPGL plotting operations.
4. When finished, close the file and, if it is not already there, COPY or RENAME it into the spooler directory.

9

9-12 Using Printers and Plotters for Generating Graphics

5. Wait for the file to be output to the plotter.

Example of Plotting to a File

You can use PLOTTER IS to send data to a file, which you can later send to an SRM plotter. The following command sequence illustrates this spooling method:

```
CREATE BDAT "/PL/Plot_file",1
PLOTTER IS "/PL/Plot_file"
FRAME
MOVE 0,0
DRAW 100,100
MOVE 0,100
DRAW 100,0
GSEND "CS2;"
GSEND "LBThis is X in the plotter's character set 2."&CHR$(3)&";"
PLOTTER IS CRT,"INTERNAL"
```

PLOTTER IS works only with BDAT files. Because the SRM 1.0 operating system's spooling works only with ASCII files, you cannot use PLOTTER IS for plotter spooling with that version of SRM.

Checking the Spooler's Status

The SRM spooler waits until the file is non-empty and closed before sending its contents to the output device. If your file is not plotted within a reasonable amount of time, you may not have closed it. You can verify that your file is ready to be plotted by cataloging the spooler directory:

```
CAT "/PL"  or 
```

The open status (OPEN STAT) of the file currently being printed or plotted is listed as locked (LOCK). Files currently being written to the spooler directory are listed as OPEN. Files that do not have a status word in the catalog are ready for plotting.

Version 2.0 (and later versions) of the SRM operating system allow BDAT files to be sent to the printing device as a byte stream. (With SRM version 1.0, only ASCII files can be used.)

Aborting Plotting In Progress

To abort an in-progress plot, use the SPOOL ABORT command from the SRM server. The system stops sending data to the output device and closes, then purges the file. For details on bringing the spooler UP and DOWN, see the description of the SPOOLER command in the "Language Reference" section of the *SRM System Manager's Guide*.

With SRM 2.0, if a plotter is taken off-line while a file is being spooled, the spooler stops and resumes when the plotter is put back on-line. No data is lost during such an interruption. SRM 1.0 also resumes plotting, but from the beginning of the file.

Dumping Graphics to a Printer Spooler

DUMP DEVICE IS allows you to direct graphics generated by DUMP GRAPHICS to a file that can be spooled. For example:

```
100 CREATE BDAT "/PL/Plot_file",1   create file before dump
110 DUMP DEVICE IS "/PL/Plot_file"
120 DUMP GRAPHICS
```

Note that you can only DUMP GRAPHICS to a BDAT or HP-UX file, and you must CREATE the file before making it the dump device. Refer to the *HP BASIC Language Reference* for more information on CREATE, DUMP DEVICE IS, and DUMP GRAPHICS.

Using a Plotter with BASIC/UX

A multi-user environment does not lend itself to dedicating a plotter to one person. Instead, multi-user systems have a plotter spooler so jobs can share the same plotter without the output from two or more jobs being interleaved. This section explains how to send your output to a plotter spooler device on HP-UX.

Prerequisites

Your HP-UX system should have a spooled plotter connected to it if more than one person will be plotting to the same plotter. If you are the System Administrator, read the section "Setting Up the LP Spooler" in the chapter "Configuring the HP-UX System" in the *HP-UX System Administrator Manual*. Since the plotter spooler is set up the same way as a printer spooler, use the same information that you used to set up the printer spooler.

Note that all spooled devices must be accessed through the server on a diskless system, and you cannot have spooled plotters on a diskless node.

Assigning a Plotter Spooler as the Output Device

The output of BASIC/UX commands can be sent as input to HP-UX commands. This feature gives you access to the HP-UX plotter spooler. For example, executing

```
PLOTTER IS "| lp -dplotter_1","HPGL"
```

assigns the destination plotter to be `lp -dplotter_1`. Note that the vertical bar symbol (`|`) is a pipe from the BASIC PLOTTER IS command to the HP-UX command `lp -dplotter_1`. The term **pipe** means that the output or result of one command is sent as the input to another command. This is the recommended way to assign a plotter from BASIC/UX. However, you can still assign the plotter as follows:

```
PLOTTER IS 705,"HPGL"
```

where 705 is the device selector for a plotter connected to the system.



Sending Graphics Output to the Plotter Spooler

The following program assigns the plotting device to be the HP-UX plotter spooler, positions the plotter pen, draws a rectangle, labels the rectangle, and sets the plotter spooler back to its default value (the CRT).

```
100 PLOTTER IS "| lp -dww_paper","HPGL",6.25,256.25,6.975,186.975 ! Assigns
110                                     ! the plotting device and sets the hard
120                                     ! clip limits of the plotter in millimeters.
130                                     !
140 MOVE 75,45 ! Positions the lower-left hand corner of the rectangle.
150 DRAW 75,70 ! Lines 150 to 180 draw a rectangle.
160 DRAW 122,70
170 DRAW 122,45
180 DRAW 75,45
190 !
200 MOVE 98,58 ! Sets the initial label position.
210 CSIZE 4 ! Sets the character size.
220 LABEL "RECTANGLE" ! Labels the rectangle.
230 !
240 PLOTTER IS CRT,"INTERNAL" ! Assigns the plotter to be the CRT.
250 END
```

Line 240 in the above program is important because the plotter will not plot your data until SCRATCH A or a new PLOTTER IS statement has been executed. When you execute either of these statements, the pipe is closed and data is sent to the spooler.

Assigning a Window as the Plotting Device

This section covers how BASIC/UX windows can be used as plotting devices. The ability to create windows and send graphics plots to them allows you to display several different graphics plots without erasing them each time you make a new one. To assign a window as a plotting device, execute a command similar to the following:

```
PLOTTER IS 601,"WINDOW"
```

This statement assigns the destination plotter to be window number 601. Note that you must first create window 601 with the CREATE WINDOW command before you can assign it as a plotting device. For information on how to do

this, read the section "Using BASIC/UX Window Commands" found in the *Using HP BASIC/UX* manual.

Sending a Graphics Plot to a BASIC/UX Window

The following program creates BASIC/UX window 601 and assigns it as the plotting device. It next draws a circle in window 601 and re-assigns the plotter to be the CRT or default window 600.

```
100     INTEGER Array(0:5),Xcoord,Ycoord
110     GRAPHICS ON
120     GESCAPE CRT,3;Array(*) ! Assign the hard clip values and row and
130                               ! column information to Array(*).
140     Xcoord=Array(2)           ! Maximum hard clip value in the X axis
150     Ycoord=Array(3)         ! Maximum hard clip value in the Y axis
160     CREATE WINDOW 601,250,350,Xcoord,Ycoord;LABEL "Window Dump" ! Create
170                               ! a window the size of default window 600.
180     PLOTTER IS 601,"WINDOW" ! Assign window 601 as the plotter.
190                               !
200     MOVE 75,45              ! Position the circle.
210     POLYGON 20,40,40,FILL ! Draw a circle.
220     PLOTTER IS CRT,"INTERNAL" ! The plotter is the CRT.
230     END
```


Using a Printer

A wide range of printers are supported by BASIC. This chapter covers the statements commonly used to communicate with external printers, including printers controlled by an HP-UX or SRM spooler.

Installing, Configuring, and Verifying Your Printer

For information about installing and configuring your printer, refer to the documentation shipped with your printer.

If you are using a BASIC Workstation (BASIC/WS) system, and once your printer is installed, verify its operation by using the “Peripheral Verification Utility” (VERIFY) described in the “Verifying and Labeling Peripherals” chapter of *Installing and Maintaining HP BASIC*. BASIC/UX does not have a VERIFY utility.

Selecting the System Printer

The PRINT statement normally directs text to the screen of the CRT.

```
PRINT "This message goes to the printer."  
PRINT 3.14159,2.71828,"String",55
```

Text may be re-directed to an external printer by using the PRINTER IS statement. The PRINTER IS statement is used to change the system printer.

- PRINTER IS 701** Selects the printer at select code 7 and address 01 as the system printer. (Recommended for single-user systems only.)
- PRINTER IS 23** Selects a Centronics-compatible parallel printer connected to the HP Parallel interface (for BASIC/WS only) on an HP 9000 Model 345, 362, 375, 380, or 382 computer. (Select code 23 is the default for the HP Parallel interface. To determine the actual select code, check the boot screen when you turn on the computer.)

Note

HP BASIC/UX does not directly support the HP Parallel interface. However, the user can use parallel peripherals on HP-UX via unnamed pipes. (For example, use **PRINTER IS "| lp"** where lp spools to the parallel printer.)

- PRINTER IS 26** Selects a Centronics-compatible parallel printer connected to the LPT1 port on a PC with the HP Measurement Coprocessor installed.
- PRINTER IS 9** For BASIC/WS or BASIC/UX, selects a serial printer connected to an RS-232 serial interface. For BASIC/DOS, selects a serial printer connected to the COM1 port on a PC with the HP Measurement Coprocessor installed. (Provided that the COM1 port is configured to select code 9 in the BLP.CON file.)
- PRINTER IS "| lp"** Selects a pipe to a spooler as the system printer. (Only works with BASIC/UX; recommended for multi-user systems. The subsequent section called "Using an HP-UX Printer Spooler" tells why.)
- PRINTER IS CRT** Returns the system printer to the computer screen. (This is the default printer.)

The expressions such as 701, 23, and CRT in the above examples are known as **device selectors**.

10-2 Using a Printer

What Are 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.

To direct the output of PRINT statements to the CRT, use the following statements:

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

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.

When more than one device can be connected to an interface, such as the internal HP-IB interface (interface select code 7), extra information is required. This extra information is the **primary address**.

Using Device Selectors to Select Printers

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

PRINTER IS 701 PRINTER IS PRT	Specifies a printer with interface select code 7 and primary address 01 (PRT is a numeric function whose value is always 701).
PRINTER IS 1407	Specifies a printer with interface select code 14 and primary address 07.
PRINTER IS 12	Specifies a printer connected to interface select code 12.
CAT TO #701	Prints a disk catalog on the printer at device selector 701.
PRINTALL IS 703	Logs error information on a printer whose select code is 7 and whose primary address is 03.
LIST #701	Lists the program in memory to a printer at 701.

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

```
PRINTER IS Hal
```

```
CAT TO #Dog
```

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

Using Control Characters and Escape Sequences

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

Control Characters

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

Typical Printer Control Characters

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

10-4 Using a Printer

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

```
PRINT CHR$(12)    print a form-feed
```

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

Escape-Code Sequences

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

For example, the HP 2631 printer is capable of printing characters in several different fonts.

```
10 PRINTER IS 701
20 Esc$=CHR$(27)           send escape before font command
30 Big$="#k1S"             command for big font
40 Regular$="#k0S"         command for regular font
50 PRINT Esc$;Big$;"Extended-Font Text"
60 PRINT Esc$;Regular$;"Back to normal."
70 PRINTER IS 1
80 END
```

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

Formatted Printing

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

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

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

Produces:

```
123456789012345678901234567890123456789
 1.1   -22.2   300000   5.1E+8
```

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

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

```
123456789012345678901234567890123
 1.1 -22.2 300000 5.1E+8
```

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

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

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

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

Produces:

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

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

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

```
123456789012345678901234567890123
                               -1.414
```

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

10-6 Using a Printer

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

Using Images

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

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

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

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

```
3.142
```

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

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

```
3.1415926536
```

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

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

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

Both PRINT statements use the image in line 100.

Numeric Image Specifiers

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

Numeric Image Specifiers

Image Specifier	Purpose
D	Replace this specifier with one digit of the number to be printed. If the digit is a leading zero, print a space. If the value is negative, the position may be used by the negative sign.
Z	Same as "D" except that leading zeros are printed.
E	Prints two digits of the exponent after printing the sequence "E+". This specifier is equal to "ESZZ". See the <i>HP BASIC Language Reference</i> 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)
R	Print the comma (European radix). (Requires IO)
*	Like Z, except that asterisks are printed instead of leading zeros. (Requires IO)

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

Examples of Numeric Image Specifiers

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

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

Multiple-Field Numeric Image Specifiers

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

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

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

```
123456789012345678901234567890123
 3.98  5.95  27.50 139.95
```

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

String Image Specifiers

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

String Image Specifiers

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

The following examples show various ways to use string specifiers.

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

```
12345678901234567890123456789
   Tom      Smith
```

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

```
12345678901234567890123456789
   John Smith
```

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

```
12345678901234567890123456789
PART NUMBER  90001234
```

Additional Image Specifiers

The following image specifiers serve a special purpose.

Additional Image Specifiers

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

For example:

```
PRINT USING "0,#"   outputs a formfeed
PRINT USING "D,X,3A,""OR NOT"",X,B,X,B,B";2,"BE",50,66,69
```

Special Considerations

If nothing prints, check if the printer is ON LINE. When the printer 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** or **Break**, check the interface cable and switch settings, then try again.

Since the printer's device selector may change, keep track of the locations in the program where a device selector is used. If most of the program's output is sent to a printer, you may wish to use the `PRINTER IS` statement at the beginning of the program and then send messages to the CRT screen by using the `OUTPUT` statement.

```
10 PRINTER IS 701
20 PRINT "Text to the printer."
30 OUTPUT 1;"Screen Message"
40 PRINT "Back to the printer."
```

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

Using Printers through the SRM Spooler

The SRM system not only provides shared access to printers and plotters, but also manages their use so that workstations never need to wait for output to be generated.

To use shared printers, you place files to be printed into a special directory where they are held until the printer is free. The system keeps track of the order in which files arrive from the workstations, and outputs them in the same order. This method is called "spooling," and the directory where the files are kept is called the "spooler directory." Spooler directories are created for the SRM controller's use when the shared peripherals are installed on the SRM system. After a file is placed in a spooler directory, the workstation is free to do other processing.

Using an SRM Spooler

Use of special SRM directories called "spooler directories" allows you to access a shared printer or plotter. Setting up a spooler directory is explained in the "Interfaces and Peripherals" chapter of the *SRM System Manager's Guide*. The examples in this section assume that the spooler directories LP (for "Line Printer") and PL (for "PLotter") have been created at the root of the SRM directory structure.

10-12 Using a Printer

SRM Printer Spooling Using PRINTER IS

You can use the PRINTER IS statement to direct data to your shared printer. The following command sequence illustrates this spooling method:

```
CREATE BDAT "/LP/Print_file",1
PRINTER IS "/LP/Print_file"
LIST
XREF
PRINTER IS CRT
```

PRINTER IS works only with BDAT files. Because the SRM 1.0 operating system's spooling works only with ASCII files, you cannot use PRINTER IS for spooling with that version of SRM.

Writing Files to SRM Spooler Directories

You may also access the printer associated with LP by placing the data to be printed in an ASCII or BDAT file in that spooler directory. For example, to list a program currently in memory, you could SAVE the program in LP as the file P1_LISTING by typing either:

```
SAVE "LP/P1_LISTING:REMOTE"  or 
or
SAVE "/LP/P1_LISTING"  or 
```

The SAVE statement creates an ASCII file. Although this is the same syntax used to save programs on a shared disk, the SRM system knows that LP is a spooler directory and prints the file as soon as possible.

Note



When used for spooling, SAVE places a file in the spooler directory. The file is printed, then purged. You may wish to save or create the file first, then use the COPY statement to place the file into the spooler directory.

Sending Program Output to a Shared SRM Printer

To spool program output to a shared printer, create an ASCII or BDAT file, assign an I/O path name to the file (which opens the file), and OUTPUT the data to that file. With BDAT files, you should ASSIGN with FORMAT ON. When the file's contents are to be printed, close the file. The following example

program segment outputs the data stored in the string array called `Data$` to an ASCII file named `PERFORMANCE`.

```

760 CREATE ASCII "/LP/PERFORMANCE",100
770 ASSIGN @Spool TO "/LP/PERFORMANCE"
780 OUTPUT @Spool;"Performance Summary"
790 OUTPUT @Spool;Data$(*)
800 ASSIGN @Spool TO * ! Initiate printing.

```

The system waits until the file is non-empty and closed before sending its contents to the output device. If your file is not printed or plotted within a reasonable amount of time, you may not have closed it. You can verify that your file is ready to be printed or plotted by cataloging the spooler directory:

```
CAT "/LP" Return or ENTER
```

The open status (`OPEN STAT`) of the file currently being printed or plotted is listed as locked (`LOCK`). Files currently being written to the spooler directory (either printer or plotter) are listed as `OPEN`. Files that do not have a status word in the catalog are ready for printing or plotting.

Version 2.0 of the SRM operating system (and later versions) allow `BDAT` files to be sent to the printing device as a byte stream. (With SRM version 1.0, only ASCII files can be used.)

Note



With the SRM 2.0 operating system, a `BDAT` file sent to the spooler is printed exactly as the byte stream sent. Unless you set up the `BDAT` file correctly, improper printer output or operation could result. Therefore, you should `ASSIGN` `BDAT` files with `FORMAT ON` before outputting data.

The spooler prints each string and numeric item on a separate line by inserting a carriage return and line feed after each item. To put several strings on one line, concatenate them into one string before using `OUTPUT` to send them to the spooler file. You may insert ASCII control characters in the data by using the `CHR$` string function.

Appearance of Output

Printed output for each file includes a one-page header, which identifies the directory path to the file, the file's name, and the date and time of the printing.

10-14 Using a Printer

To cause the printer to skip the paper perforation after printing a page (60 lines), prefix your file name with "FF". For example:

```
SAVE "/LP/FF_MYTEXT" Return or ENTER
```

Using an HP-UX Printer Spooler (BASIC/UX Only)

A multi-user environment does not lend itself to a printer being dedicated to one person. Instead, multi-user systems have a "printer spooler" so jobs to be output can share the same printer without one person's output being mixed (interleaved) with someone else's output. This section explains how to send printer output to a spooler set up on HP-UX.

Prerequisites

The HP-UX system that you are running BASIC/UX on needs to have a spooled printer connected to it. If you are the System Administrator, read the section "Setting Up the LP Spooler" in the chapter "Configuring the HP-UX System" in the *HP-UX System Administrator Manual*. If you are not the system administrator, have your System Administrator set the spooler up for you.

A Simple Example

Note



The term "pipe" is a mechanism that uses the output or result of one command as the input to another command. For more information on "pipes," refer to *HP BASIC 6.2 Advanced Programming Techniques*.

```
PRINTER IS "| lp"
```

Assigns the destination printer to be lp (the line printer spooler on HP-UX). Note that the vertical bar symbol (|) is a pipe from the BASIC PRINTER IS command to the HP-UX lp command.

<code>PRINT "To the printer spooler."</code>	Sends characters to the spooler <i>file</i> (but not to the printer yet).
<code>PRINTER IS CRT</code>	Re-assigns the system printer to be the CRT, and "closes" the spooler file. Once the spooler file is closed, the spooler automatically prints it.
<code>PRINT "Back to the CRT."</code>	Prints message on CRT.

You can also use pipes with the `PRINTALL IS` statement. (This command causes all traced messages and input typed at the keyboard to be sent to the printer spooler.) It is a good way to keep a written record of what you have entered at the keyboard and interactions during debug traces. For more information, read the section "Tracing" in chapter 13.

Using the Clock and Timers

All Series 200 and 300 computers and the HP Measurement Coprocessor provide a volatile real-time clock (the “BASIC clock”) that you can set and read to monitor the time of day and date. There are also timers that can interrupt a BASIC program. This chapter describes using the clock- and timer-related functions and statements.

BASIC CLOCK Binary

With the BASIC/WS system, many of the statements described in this chapter require the CLOCK binary. This is also true with BASIC/DOS for the HP Measurement Coprocessor. The *HP BASIC Language Reference* lists which binary (if any) is required to use each statement and function. (The BASIC/UX system is not partitioned into separately loadable binaries.)

Do You Have a Battery-Powered Clock?

All Series 300 computers have a non-volatile clock (a non-volatile clock is powered by a battery so that it keeps time even when the computer is turned off). However, some Series 200 computers do not have this type of clock. All HP Vectra PCs and most AT-compatible PCs also have a battery-powered clock, which is used in conjunction with the HP Measurement Coprocessor.

If you don't have a non-volatile clock, you may need to determine whether or not the real-time clock is set to the proper time.

Initial Clock Value

When you initially boot the BASIC system, the real-time clock is set to one of these values:

- With Series 300 computers, the clock value is read from the non-volatile clock and placed into the real-time (volatile) clock.
- With the HP Measurement Coprocessor, the clock value is read from the PC's non-volatile clock, if present, and placed into the real-time clock. (This occurs when you boot the measurement coprocessor, not the PC.)
- Each BASIC/UX process has its own "local" clock value, which is an offset from the value of the HP-UX system clock. (The default value of this offset is 0, which means that BASIC/UX's "local" clock is initially set to match the HP-UX system's clock.)
- With Series 200 computers which have the 98270 Powerfail Option installed, the real-time (volatile) clock time is set to the value of the non-volatile clock.
- With computers on the Shared Resource Management (SRM) system that *don't* have a non-volatile clock, the clock value is taken from the SRM system. (This occurs provided the SRM and DCOMM binaries are loaded.)
- If the computer does not have a non-volatile clock (and is not connected to an SRM system), the time is set to 12:00:00 a.m. (midnight), March 1, 1900.

Clock Range and Accuracy

The range of Series 200 volatile and non-volatile clocks is March 1, 1900 through August 4, 2079. The Series 300 volatile and non-volatile clocks both have a lower limit of March 1, 1900. However, the upper limit of the volatile clock is August 4, 2079, while the upper limit of the non-volatile clock is February 29, 2000.

The volatile real-time clocks provide an accuracy of ± 5 seconds per day. The Series 200 battery-backed "powerfail" (98270) clock maintains time to within ± 2.5 seconds per day. The Series 300 battery-backed clock maintains time to within ± 5 seconds per day.

11-2 Using the Clock and Timers

The range of the HP Measurement Coprocessor volatile clock is March 1, 1900 through August 4, 2079. The range and accuracy of the non-volatile clock (if present) depends on the PC in which the measurement coprocessor is installed.

Reading the Clock

Internally, the 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 the 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)
```

Prints: 1 Mar 1900

This is the default power-up date for machines without the battery-backed real-time clock.

The TIME\$ function returns the time of day.

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

Prints: 00:05:27



11 Setting the Time, Date, and Time Zone

BASIC/WS and HP-UX Clock Compatibility

If you are using a BASIC “Workstation” system (BASIC/WS) and you will be sharing an HFS disk with an HP-UX system, then you will need to use the TIMEZONE IS statement *before* setting the clock. See the *HP BASIC Language Reference* entry for TIMEZONE IS for instructions. (Note that with BASIC/UX, you will *not* need to use TIMEZONE IS.)

Setting the Time and Date

The SET TIMEDATE statement is used to set the clock:

```
SET TIMEDATE DATE("1 Oct 1988") + TIME("08:37:30")
```

The BASIC/UX “Local” Clock

The time and date functions are set the same on a BASIC/UX system as on a BASIC/WS system. However, note that the time and date you are changing are just “offsets” from the HP-UX clock that the BASIC/UX process uses. For instance, if you set the clock to be two hours ahead of what it is now, your “local” clock value will change, but the HP-UX system clock will not.

If it is your intent to change the time of the HP-UX system that BASIC/UX is running on, you need to be logged in as *root* (the system administrator) and use the HP-UX system commands for changing the time, date, and time zone. See the *HP-UX System Administrator Manual* for instructions.

Clock Time Format

To minimize the space required to store the date and time, and yet insure a unique value for each point in time, both time and date are combined as a single real number. This value is the Julian date multiplied by the number of seconds in a day. By recalling that there are 86400 seconds in a day, the date and time of day can be extracted from TIMEDATE by the following simple algorithms.

11-4 Using the Clock and Timers

TIMEDATE MOD 86400 *returns the time of day*
TIMEDATE DIV 86400 *returns the Julian date*

The time of day is expressed in seconds past midnight and is easily divided into hours, minutes, and seconds. The Julian date requires a bit more processing to extract the month, day, and year but this method insures a unique value for each day over the entire range of the clock.

Year	Clock Value	Hours	Seconds
1900	2.086578144E+11	1	3600
1910	2.089733472E+11	2	7200
1920	2.092888800E+11	3	10800
1930	2.096044992E+11	4	14400
1940	2.099200320E+11	5	18000
1950	2.102356512E+11	6	21600
1960	2.105511840E+11	7	25200
1970	2.108668032E+11	8	28800
1980	2.111823360E+11	9	32400
1990	2.114979552E+11	10	36000
2000	2.118134880E+11	11	39600
2010	2.121291072E+11	12	43200
2020	2.124446400E+11	13	46800
2030	2.127602592E+11	14	50400
2040	2.130757920E+11	15	54000
2050	2.133914112E+11	16	57600
2060	2.137069440E+11	17	61200
2070	2.140225632E+11	18	64800
2080	2.143380960E+11	19	68400
		20	72000
		21	75600
		22	79200
		23	82800
		24	86400

Setting Only 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 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.

The following program contains the routines to set and display the time of day. The routines are written as user-defined functions that may be appended to your program. Once appended to a program, the routines duplicate the `TIME` and `TIME$` functions available with `CLOCK`. The formatted time can then be displayed by the following statement.

```
PRINT FNTIME$(TIMEDATE)
```

Prints: 15:31:05

Given the clock's value, the `FNTIME$` function returns the time of day in 24 hour format (HH:MM:SS). The `FNTIME` function converts the time of day to seconds and is used to set the clock.

11-6 Using the Clock and Timers

```

10 Show_time:DISP FNTIME$(TIMEDATE)
20   GOTO Show_time
30   END
40   !
50   ! While the program is running, type:
60   ! SET TIME FNTIME("11:5:30")
70   ! then press <EXECUTE> to show the new time.
80   !
90   !*****
100  !
110  DEF FNTIME$(Now) ! Given 'SECONDS' Return 'hh:mm:ss'
120  !
130  Now=INT(Now) MOD 86400
140  H=Now DIV 3600
150  M=Now MOD 3600 DIV 60
160  S=Now MOD 60
170  OUTPUT T$ USING "#,ZZ,K";H,":",M,":",S
180  RETURN T$
190  FNEND
200  !
210  DEF FNTIME(T$) ! Given 'hh:mm:ss' Return 'SECONDS'
220  !
230  ON ERROR GOTO Err
240  ENTER T$;H,M,S
250  RETURN (3600*H+60*M+S) MOD 86400
260 Err:OFF ERROR
270  RETURN TIMEDATE MOD 86400
280  FNEND

```

After entering the program, follow the instructions at the beginning of the program to verify correct operation. Store this program in a file named "FUNTIME". The functions can be extracted from this program and appended to other programs by the LOADSUB statement.

Note that the FNTIME function requires hours, minutes, and seconds, while the TIME function only requires hours and minutes.

Setting Only 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 and FNDate functions convert the accompanying string (or string expression) into the numeric value needed to set the clock. To read the clock, the DATE\$ and FNDate\$ functions format 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

Programs that need to run without CLOCK can use the following user-defined functions appended to the end of the program. These functions simulate the DATE and DATE\$ keywords available in CLOCK. The algorithm is valid over the entire range of the clock.

Note the following functions are restricted to values the clock will accept, the DATE and DATE\$ functions available with CLOCK allow a much wider range of values (including negative years).

```
10 Show_date:  DISP FNDate$(TIMEDATE)
20             GOTO Show_date
30             END
40           !
50           ! While the program is running, type:
60           ! SET TIMEDATE FNDATE("1 JAN 82")  <EXECUTE>
70           !
```

11-8 Using the Clock and Timers

```

80 !*****
90 !
100 DEF FNDate$(Seconds) ! Given 'SECONDS' Return 'dd mmm yyyy'
110 !
120 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
130 DIM Month$(1:12)[3]
140 READ Month$(*)
150 !
160 Julian=Seconds DIV 86400-1721119
170 Year=(4*Julian-1) DIV 146097
180 Julian=(4*Julian-1) MOD 146097
190 Day=Julian DIV 4
200 Julian=(4*Day+3) DIV 1461
210 Day=(4*Day+3) MOD 1461
220 Day=(Day+4) DIV 4
230 Month=(5*Day-3) DIV 153 ! Month
240 Day=(5*Day-3) MOD 153
250 Day=(Day+5) DIV 5 ! Day
260 Year=100*Year+Julian ! Year
270 IF Month<10 THEN
280 Month=Month+3
290 ELSE
300 Month=Month-9
310 Year=Year+1
320 END IF
330 OUTPUT D$ USING "#,ZZ,X,3A,X,4Z";Day,Month$(Month),Year
340 RETURN D$
350 FNEND
360 !
370 DEF FNDate(Dmy$) ! Given 'dd mmm yyyy' Return 'SECONDS'
380 !
390 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
400 DIM Month$(1:12)[3]
410 READ Month$(*)
420 !
430 ON ERROR GOTO Err
440 I$=Dmy$&" "
450 ENTER I$ USING "DD,4A,5D";Day,M$,Year
460 IF Year<100 THEN Year=Year+1900
470 FOR I=1 TO 12
480 IF POS(M$,Month$(I)) THEN Month=I
490 NEXT I
500 IF Month=0 THEN Err
510 IF Month>2 THEN
520 Month=Month-3

```



```

530     ELSE
540         Month=Month+9
550         Year=Year-1
560     END IF
570     Century=Year DIV 100
580     Remainder=Year MOD 100
590     Julian=146097*Century DIV 4+1461*Remainder DIV 4+(153*Month+2) DIV
5+Day+1721119
600     Julian=Julian*86400
610     IF Julian<2.08662912E+11 OR Julian>=2.143252224E+11 THEN Err
620     RETURN Julian ! Return Julian date in SECONDS
630 Err:OFF ERROR      ! ERROR in input.
640     RETURN TIMEDATE ! Return current date.
650     FNEED

```

Store the program in a file named "FUNDATE". Later the functions can be appended to other programs by the LOADSUB statement.

The functions `FNDate$` and `FNDate` format the date as "DD MMM YYYY", where DD is the day of the month, MMM is the first three letters of the month, and YYYY is the year. The function `FNDate` will accept the last two digits of the year. See line 460. Note that the `FNDate` function requires two digits for the day, while the `DATE` function does not.

Different formats require only slight modification. By changing the following lines, the date is formatted as "MM/DD/YYYY".

```

330 OUTPUT D$ USING "#,2D,A,2D,A,2D";Month;"/";Day;"/";Year
450 ENTER I$ USING "#,ZZ,K";Month;Day;Year

```

European date format is obtained by swapping the month and day in the above statements. When changing the format, be sure to switch both functions.

If the all numeric format is chosen, delete the three lines in each function that load the array with the month mnemonics.

Using Clock Functions and Example Programs

11

The following statements summarize setting and displaying the clock.

```
SET TIMEDATE FNDate("12 DEC 1981") + FNTime("13:44:15")

SET TIME FNTime("8:30:00")

PRINT FNTime$(TIMEDATE)

DISP FNDate$(TIMEDATE)
```

It is important to note that SET TIMEDATE expects a date and time while the DATE function and the user-defined function FNDate return only a date. This effectively sets the clock to midnight of the date specified.

To keep the functions short, minimal parameter checking is performed. Additional checking may be incorporated within the functions or within the calling context. If FNDate or FNTime cannot correctly decode the input, the current value of the clock is returned.

The date and time functions can be used with the following program shell to provide a "friendly" interface to the clock.

```
10  ! PROGRAM SHELL FOR SETTING TIME AND DATE.
20  !
30  ! REQUIRES THE TIME AND DATE FUNCTIONS.
40  !
50  DIM Day$(0:6)[9]
60  DATA Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday
70  READ Day$(*)
80  !
90  ON ERROR GOTO Nofun      ! Test if functions
100 Dmy$=FNDate$(TIMEDATE) ! have been loaded
110 Hms$=FNTime$(TIMEDATE)
120 OFF ERROR
130 Main:                    ! Get NEW date
140 CLEAR SCREEN
150 F$=CHR$(255)&CHR$(72)
160 !
170 PRINT TABXY(1,14);"Enter the date, and press CONTINUE."
180 OUTPUT 2 USING "#,11A,2A";Dmy$,F$
190 !
200 INPUT Dmy$              ! WAIT for INPUT
210 !
```

```

220 ENTER Dmy$ USING "2D,4A,5D";D,M$,Y
230 CLEAR SCREEN
240 !
250 PRINT TABXY(1,14);"Enter the time of day and press CONTINUE"
260 OUTPUT 2 USING "#,11A,2A";Hms$,F$
270 INPUT Hms$
280 ENTER Dmy$ USING "2D,4A,5D";D,M$,Y
290 !
300 SET TIMEDATE FNDate(Dmy$)+FNTime(Hms$)
310 !
320 CLEAR SCREEN
330 W=(TIMEDATE DIV 86400) MOD 7 ! Day of week
340 PRINT TABXY(1,1);"The clock has been set to:"
350 PRINT TABXY(1,3);Day$(W);" ";Dmy$;" ";FNTime$(TIMEDATE)
360 GOTO Quit
370 !
380 ! ***** SUBROUTINES *****
390 !
400 Nofun:PRINT "The TIME & DATE FUNCTIONS must be appended,"
410 PRINT "(via LOADSUB) before program will work."
420 Quit:END
430 !
440 ! ***** FUNCTIONS *****
450 !
460 ! append time and date functions here

```

The program tests to see if the functions have been loaded by trying to use them. If they are not loaded the program ends with an error message. With the CLOCK binary, this program can still be used. Replace the calls to the user-defined functions with the appropriate keywords. The error trapping can then be deleted.

To append the user-defined functions, execute the following statements while the demonstration program is in memory.

```

LOADSUB ALL FROM "FUNDATE"
LOADSUB ALL FROM "FUNTIME"

```

Examine the program to be sure the functions have been loaded.

The program will prompt for the date and time, then set the clock accordingly. A program such as this may be used as the system start up program for applications requiring the date or time.

11-12 Using the Clock and Timers

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). See the previous program for an example of using this routine.

Days Between Two Dates

The number of days between two dates is easily calculated as the following program demonstrates.

```

10   ! Days between two dates
20   INPUT "ENTER THE FIRST DATE (DD MMM YYYY)",D1$
30   INPUT "ENTER THE SECOND DATE (DD MMM YYYY)",D2$
40   Days=(DATE(D2$)-DATE(D1$)) DIV 86400
50   DISP Days;"days between ";D1$;" and ";D2$;"'"
60   END

```

Interval Timing

Timing a single event of short duration is quite simple.

```

10   T0=TIMEDATE           ! Start
20   FOR J=1 TO 5555
30   !
40   NEXT J
50   T1=TIMEDATE           ! Finish
60   !
70   PRINT "It took";DROUND(T1-T0,3);"seconds"
80   END

```

Programs can and should be written so that they do not change the setting of the clock. A short program, which simulates a stopwatch, allows interval timing without changing the clock.

```

10   ! Program: STOPWATCH
20   ! Interval timing without changing the clock
30   ON KEY 5 LABEL " START " GOTO Start
40   ON KEY 6 LABEL " STOP  " GOTO Hold
50   ON KEY 7 LABEL " RESET " GOTO Reset
60   ON KEY 8 LABEL " LAP  " GOSUB Lap
70   !

```

```

80 Reset:PRINT CHR$(12)           ! form-feed
90  H=0                           ! Set all
100 M=0                           ! to
110 S=0                           ! zero.
120  !
130 Hold:DISP TAB(9);H;";";M;";";S ! Wait til
140  GOTO Hold                   ! keypress
150  !
160 Lap:PRINT H;";";M;";";S      ! Print lap
170  RETURN
180  !
190 Start:Z=3600*H+60*M+S-TIMEDATE ! Elapsed-
200 Loop:T=(TIMEDATE+Z) MOD 86400 ! time
210  T=INT(T*100)/100           ! .01 sec.
220  H=T DIV 3600               ! Hours
230  M=T MOD 3600 DIV 60       ! Minutes
240  S=T MOD 60                ! Seconds
250  DISP TAB(9);H;";";M;";";S ! Show time
260  GOTO Loop                 ! Do again
270  END

```

Branching on Clock Events

Several additional branching statements, available with `CLOCK`, allow end-of-statement branches to be triggered according to the real-time clock's 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.

11-14 Using the Clock and Timers

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 internal real-time clock. Care should be taken 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.

Clock Resolution

The clock resolution is different for two implementations of BASIC:

- 10 milliseconds for BASIC/WS and BASIC/DOS.
- 20 milliseconds for BASIC/UX (timing accuracy depends on the system load).

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:

```

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).

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, 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

11-16 Using the Clock and Timers

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 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. The following program will illustrate this effect.

```

10   COM Counter
20   Counter=0
30   GINIT
40   GRID 1,1           ! Fill graphics raster with grid.
50   DISP Counter
60   ON CYCLE 2 CALL Flash ! Flash graphics every 2 seconds.
70 W: GOTO W
80   END
90   !----- SUB to flash graphics raster -----
100  SUB Flash
110   COM Counter
120   GRAPHICS ON
130   Counter=Counter+1
140   DISP Counter
150   IF Counter=5 THEN           ! Change CYCLE value during fifth CALL.
160   ON CYCLE .1,2 CALL Quit ! New value (.1) will replace old (2).

```

11-18 Using the Clock and Timers

```

170                                     ! Flash will end before Quit gets called.
180     END IF
190     GRAPHICS OFF
200     SUBEND
210     !----- SUB that won't get called -----
220     SUB Quit
230         PRINT "PROGRAM HAS STOPPED"
240         STOP
250     SUBEND

```

The program starts out by flashing the graphics raster on and off every two seconds. When the subprogram's ON CYCLE statement is activated during the fifth call to the subprogram, the new value (0.1 second) replaces the old value (2.0 seconds) and the program begins flashing the graphics raster at the new rate. Note that the branch to the second subprogram (Quit) is not executed because the first subprogram is finished before the specified time. To see the second subprogram execute, insert the following line.

```

191     WAIT 1

```

The delay caused by the WAIT statement allows the subprogram's ON CYCLE statement to branch to the second subprogram and stop execution.

If an active branch defined in the main program is canceled in a subprogram (by OFF TIME, OFF DELAY, or OFF CYCLE) any branch missed during the execution of the subprogram will be lost. When the context containing the original statement is restored, the branch will be reactivated and processing will continue as if no branch was missed.

```

10     ON DELAY 1 GOTO Done           ! GOTO "Done" in one second.
20     CALL Busy                     ! Call to "Busy" takes two seconds.
30     !
40     PRINT "THIS WON'T BE PRINTED UNLESS BRANCH IS CANCELED BY THE SUB"
50     !
60 Done:PRINT "THIS LINE WILL BE PRINTED EVERY TIME"
70     END
80     ! -----
90     SUB Busy
100    WAIT 2
110    ! OFF DELAY ! RUN then remove the "!" on this line and RUN again.
120    SUBEND

```

By removing the comment symbol (!) from the beginning of line 110, the OFF DELAY statement will be executed causing any branch that has already been logged to be canceled and allow line 40 to be printed.

Since branches only occur at the end of a line, no branch can be taken during an INPUT or LINPUT statement. The following program shows a method of monitoring the keyboard without preventing branches to be taken.

```

10  ON KBD GOTO Yes      ! If key is pressed go get new value.
20  ON DELAY 3 GOTO Gone ! If no keypress in 3 seconds use defaults
30  DISP "PRESS A KEY"
40 W: GOTO W            ! Wait here until keypress or end of delay.
50  !
60 Yes:OFF DELAY       ! Someone is there.
70  OFF KBD
80  LINPUT "NEW VALUE?",A$
90  DISP "USING",A$
100 GOTO More
110 !
120 Gone:DISP          ! Nobody there.
130 DISP "USING DEFAULTS"
140 !
150 More:WAIT 2
160 DISP "program continues.... "
170 END

```

The program waits a few seconds for a response. Processing continues with default values if no key is pressed. Pressing a key will cause the program to accept the new information.

Handling Errors

Most programs are subject to errors at run time, even if all the typographical/syntactical errors have been removed in the process of entering the program into the computer in the first place. This chapter describes how BASIC programs can respond to these errors, and shows how to write programs that attempt to either correct the problem or direct the program user to take some sort of corrective action. For a complete listing of BASIC error messages, refer to “Error Messages” in the *HP BASIC Language Reference* manual.

Overview of Error Responses

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

1. Try to prevent the error from happening in the first place (by communicating clearly with the program user, by using range-checking routines, and so forth).
2. Once an error occurs, try to recover from it and continue execution (this involves the BASIC program trapping and correcting errors).
3. Do nothing—let the system stop the program when an error happens.

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

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement, and the nature of HP Series 200/300 BASIC is such that this is often a feasible choice. Upon encountering a run-time error, the computer will pause program execution and display a message giving the error number and the line in which the error happened, and the programmer can

then examine the program in light of this information and fix things up. The key word here is “programmer.” If the person running the program is also the person who wrote the program, this approach works fine. If the person running the program did not write it, or worse yet, does not know how to program, some attempt should be made to prevent errors from happening in the first place, or to recover from errors and continue running.

Anticipating Operator Errors

When a programmer writes a program, he or she knows exactly what the program is expected to do, and what kinds of inputs make sense for the problem. Given this viewpoint, there is a strong tendency for the programmer not to take into account the possibility that other people using the program might *not* understand the boundary conditions. A programmer has 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. If the programmer’s outlook is noble, he or she will try to save the user from needless anguish and frustration. Even if the programmer’s outlook is less altruistic, he or she will try to keep from getting involved in future support problems. In either case, an effort must be made to make the program more resistant to errors.

Boundary Conditions

A classic example of anticipating an operator error is the “division by zero” situation. An INPUT statement is used to get the value for a variable, and the variable is used as a divisor later in the program. If the operator should happen to enter a zero, accidentally or intentionally, the program pauses with an error 31. It is far better to be watching for an out-of-range input and respond gracefully. 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

```

12

Consider another simple example of giving a user the choice of six colors for a certain bar graph. It might be preferable to have the user pick a number corresponding to the color he wished to choose instead of having to type in up to six characters. In this case, the program wouldn't have to check for each number, but rather it could use the logical comparators to check for an entire range:

```

4030 CLEAR SCREEN
4040 DATA GREEN,BLUE,RED,YELLOW,PURPLE,PINK
4050 ALLOCATE Colors$(1:6)[6]
4060 READ Colors$(*)
4070 FOR I=1 TO 6
4080     PRINT USING "DD,X,K";I,Colors$(I)
4090 NEXT I
4100 Ask: INPUT "Pick the number of a color",I
4110 IF I>=1 AND I<=6 THEN Valid_Color
4140 BEEP
4150 DISP "Invalid answer -- ";
4160 WAIT 1
4170 GOTO Ask

```

The above example needs a little extra safeguarding. I, the variable being input, should be declared to be an integer, since the only valid inputs are 1, 2, 3, 4, 5, and 6. An answer like "pick the 3.14th color listed" does not make sense.

Real number boundaries are tested for in a manner similar to that of integers:

```

7010 INPUT "Enter the waveform's frequency (in KHz)",Frequency
7020 IF Frequency<=0 THEN 7010
7030 INPUT "Enter the amplitude (0-10 volts)",Amplitude
7040 IF Amplitude<0 OR Amplitude>10 THEN 7030
7050 INPUT "Enter the phase angle (in degrees)",Angle
7060 IF Angle<0 OR Angle>180 THEN 7050
7070 Angle=Angle*PI/180

```

REAL and COMPLEX Numbers and Comparisons

A word of caution is in order about the use of the = comparator in conjunction with REAL and COMPLEX (full-precision) numbers. Numbers on this computer are stored in a binary form, which means that the information stored is not guaranteed to be an exact representation of a decimal number—but it will be very close! What this means is that a program should not use the = comparator in an IF statement where the comparison is being performed on REAL or COMPLEX numbers. The comparison will yield a 'false' or '0' value if the two are different by even one bit, even though the two numbers might really be equal for all practical purposes.

There are two ways around this problem. The first is to try to state the comparison in terms of the <= or >= comparators.

If it is necessary to do an equality comparison with a pair of REAL numbers, then the second method must be used. This involves picking an error tolerance for how close to being equal the two numbers can be to satisfy the test.

So if the difference between two REAL numbers X1 and X2 is less than or equal to a tolerance T0, we'll say that X1 and X2 are "equal" to each other for all practical purposes. The value of T0 will depend upon the application, and must be chosen with care.

For an example, assume that we've picked a tolerance of 10^{-12} for comparing two real numbers for equality. The proper way to compare the two numbers would be:

```
950 IF ABS(X1-X2)<=1E-12 THEN Numbers_equal
960 ! Otherwise they're not equal
```

Another technique for comparing two REAL or COMPLEX values is to use the DROUND function. This is especially suited to applications where the data is known to have a certain number of significant digits. For more details on binary representations of decimal numbers, refer to the "Numeric Computation" chapter.

Note that >=, <=, and DROUND *do not* work with COMPLEX numbers, but you can compare real parts and imaginary parts. For example, comparing two COMPLEX values for equality would require something like this:

```
IF (ABS(REAL(C1)-REAL(C2)) <= 1E-12) AND
   (ABS(IMAG(C1)-IMAG(C2)) <= 1E-12) THEN ...
```

12-4 Handling Errors

Trapping Errors with BASIC Programs

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

12

Setting Up Error Service Routines

(ON/OFF ERROR)

The ON ERROR statement sets up a branching condition which will be taken any time a recoverable error is encountered at run time. Here are some example statements (further examples of each type of branch—GOSUB, GOTO, etc.—are given in subsequent sections).

```
100 ON ERROR GOSUB Fix_it      400 Fix_it: ! Solve problem.  
                               .  
                               .  
                               530 RETURN ! If GOSUB used.
```

```
100 ON ERROR GOTO Fix_it  
100 ON ERROR RECOVER Fix_it   400 Fix_it: ! Solve problem.
```

```
200 ON ERROR CALL Fix_it_sub  800 SUB Fix_it_sub  
                               .  
                               .  
                               950 SUBEND
```



Choosing a Branch Type

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

- Using GOSUB indicates that you want to return to the statement that caused the error (RETURN) or to the one following the statement that caused the error (ERROR RETURN) when finished with your attempt to correct the error's cause.

- GOTO, on the other hand, may indicate that you do not want to re-attempt the operation after attempting to correct the source of the error.

Scope of Error Trapping and Recovery

12

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.

ON ERROR Execution at Run-Time

When an ON ERROR statement is executed, the BASIC system will make sure that the specified line or subprogram exists in memory before the program will proceed.

- If GOTO, GOSUB, or RECOVER is specified, then the *line identifier* must exist in the current context (at pre-run).
- If CALL is used, then the specified *subprogram* must currently be in memory (at run-time).

In either case, if the system can't find the given line, error 49 is reported.

ON ERROR Priority

ON ERROR has a priority of 17, which means that it will *always* take priority over any other ON EVENT branch, since the highest user-specifiable priority is 15.

Disabling Error Trapping (OFF ERROR)

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

The DISABLE statement has no effect on ON ERROR branching.

Determining Error Number and Location (ERRN, ERRLN, ERRL, ERRDS, ERRM\$)

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

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

```
130     PAUSE
140 END IF
```

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

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

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

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

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

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

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

```
1140 DISP ERRL(710)

910 IF ERRL(Compute) THEN Fix_compute
```

12

ERRDS returns the **device selector** of the device which was involved in the last error. For instance:

```
IF ERRDS=12 THEN GOSUB Fix_gpio
```

Note that this function is *only* updated when an error that involves an interface or device occurs; otherwise, it remains unchanged until another error involving a device selector occurs. Therefore, if the last error did not involve a device, then the value returned by ERRDS may be irrelevant to the current situation.

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

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

```
ERROR 1 in 10 Configuration error
```

A Closer Look at ON ERROR GOSUB

The ON ERROR GOSUB statement is used when you want to return to the program line where the error occurred. You have two choices of returning:

- RETURN returns program control back to *the line that caused the error*, thus indicating that you have corrected/resolved the error condition and want to *re-execute* this line.
- ERROR RETURN returns program control to the line *following* the line that caused the error, thus indicating that you have taken alternative action in the subroutine and *do not want to re-execute* the line that initially caused the error.

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

12-8 Handling Errors

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 (17) until the RETURN statement is executed, at which point the system priority is restored to the value it was when the error happened.

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

12

Note

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



A Closer Look At ON ERROR GOTO

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

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

One advantage of ON ERROR GOTO is that you can use another ON ERROR statement in the service routine (which you cannot use with ON ERROR GOSUB). This technique, however, requires that you re-establish the original error service routine after correcting any errors (by re-executing the original

ON ERROR GOTO statement). The disadvantage is that more programming may be necessary in order to resume execution at the appropriate point after each error service.

12

```
10  RESTORE
20  PRINT
30  PRINT
40  PRINT "Coefficients of quadratic equation A"
50  DATA 0,0,0
60  READ A,B,C
70  Maxreal=1.79769313486231E+308
80  Overflow=0
90  Coefficients:  !
100 INPUT "A?",A
110 IF A=0 THEN
120   DISP "Must be quadratic"
130   WAIT .5
140   GOTO Coefficients
150 END IF
160 PRINT "A=";A
170 INPUT "B?",B
180 PRINT "B=";B
190 INPUT "C?",C
200 PRINT "C=";C
210 Compute_roots:  !
220 ON ERROR GOTO Esr
230 Imaginary=0
240 Part1=-B/2.*A
250 Part2=SQR(B*B-4*A*C)/2.*A
260 IF NOT Imaginary THEN
270   Root1=Part1+Part2
280   Root2=Part1-Part2
290 END IF
300 OFF ERROR
310 Print_roots:  !
320 IF Imaginary=0 THEN
330   PRINT "Root 1 =";Root1
340   PRINT "Root 2 =";Root2
350 ELSE
360   PRINT "Root 1 =";Part1;" +";Part2;" i"
370   PRINT "Root 2 =";Part1;" -";Part2;" i"
380 END IF
390 IF Overflow THEN PRINT "OVERFLOW"
400 STOP
410 Esr:  !
```

12-10 Handling Errors

```

420 IF ERRN=30 THEN      ! SQRT of negative number
430   Part2=SQRT(ABS(B*B-4*A*C))/2*A
440   Imaginary=1
450   Branch=1
460   GOTO 270
470 ELSE
480   IF ERRN=22 THEN  ! REAL overflow
490     Overflow=1
500     SELECT ERRLN
510     CASE 240
520       Part1=SGN(B)*SGN(A)*Maxreal
530       Branch=2
540     CASE 250
550       Part2=Maxreal
560       Branch=3
580     CASE 270
590       Root1=Maxreal*SGN(Part1)
600       Branch=4
620     CASE 280
630       Root2=Maxreal*SGN(Part1)
640       Branch=5
660       PRINT "Unexpected overflow"
670       Branch=6
680     CASE ELSE
690       DISP "Unexpected error";ERRN
700       Branch=6
710     END SELECT
720   END IF
730 END IF
740 ON Branch GOTO 270,250,260,280,290,10
750 END

```

12

A Closer Look At ON ERROR CALL

ON ERROR CALL is global, meaning once it is activated, the specified subprogram will be called immediately whenever an error is encountered, *regardless of the current context*. System priority is set to level 17 inside the subprogram, and remains that way until the SUBEXIT is executed, at which time the system priority will be restored to the value it was when the error happened.

As with ON ERROR GOSUB, you will generally use the ON ERROR CALL statement when you want to return to the program where the error occurred. You have two choices of return destinations:

- SUBEXIT sends program control back to *the line that caused the error*, thus indicating that you have corrected the cause of the problem and want to *re-execute* this line.
- ERROR SUBEXIT sends program control to the line *following* the line that caused the error, thus indicating that you have taken an alternative action and do *not* want to re-execute the line that initially caused the error.

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

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



Cannot Pass Parameters Using ON ERROR CALL

Bear in mind 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.

Using ERRLN and ERRL in Subprograms

You can use the ERRLN function in any context, and it returns the line number of the most recent error. However, the ERRL function will not work in a different environment than the one in which the ON ERROR statement is declared. For instance, the following two statements will only work in the context in which the specified lines are defined:

```
100 IF ERRL(40) THEN GOTO Fix40
100 IF ERRL(Line_label) THEN Fix_line_label
```

The line identifier argument in ERRL will be modified properly when the program is renumbered (such as explicitly by REN or implicitly by GET);

12-12 Handling Errors

however, that is not true of expressions used in comparisons with the value returned by the ERRLN function.

So when using an ON ERROR CALL, you should set things up in such a manner that the line number either doesn't matter, or can be guaranteed to always be the same one when the error occurs. This can be accomplished by declaring the ON ERROR immediately before the line in question, and immediately using OFF ERROR after it.

12

```
5010 ON ERROR CALL Fix_disc
5020 ASSIGN @File TO "Data_file"
5030 OFF ERROR
.
.
.
7020 SUB Fix_disc
7030 SELECT ERRN
7040 CASE 80
7050     DISP "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
```

A Closer Look At 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.

12

```
.  
. .  
3250 ON ERROR RECOVER Give_up  
3260 CALL Model_universe  
3270 DISP "Successfully completed"  
3280 STOP  
3290 Give_up: DISP "Failure ";ERRN  
3300 END  
. .  
. .  
. .
```

Simulating Errors (CAUSE ERROR)

Since it is not always convenient to set up the conditions that cause errors, this BASIC system has a simple way of programmatically simulating errors. The following statement does this:

```
CAUSE ERROR Error_number
```

The parameter `Error_number` is the number of the error that you want to simulate (error numbers in the range 1001 through 1080 have special significance, as described later in this section.) Thus, `CAUSE ERROR` is useful in testing and verifying your error trapping routines.

The effects of this statement are the same as if the error were caused by real error conditions:

- The `ERRN` function still returns the error number (in this case, it is the value that you specified in the `CAUSE ERROR` statement).
- The `ERRM$` function still returns the text of the corresponding error message (if the `ERR` binary is present).
- The `ERRLN` function still returns the line number at which the error occurred (in this case, the line number of the `CAUSE ERROR` statement).

12-14 Handling Errors

- The ERRL function still returns a 1 when its argument is the line at which the error occurred.

Note, however, that CAUSE ERROR does *not* change ERRDS.

If CAUSE ERROR is executed from the keyboard, the appropriate error message will be reported, but none of the *program-related* error conditions are affected. (This is also true of other keyboard-related errors.)

12

Example of Simulating an Error

Here is an example of modifying one of the preceding examples to simulate an error. (Note that the original statement has been “commented out” so that it will be easy to put back in after the testing is finished.)

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

The error-trapping subroutine can then be tested to verify that it properly traps error 30. After this verification, you may want to modify the CAUSE ERROR line to simulate other errors that could possibly occur at that point in the program. (In this example, it is not necessary since all other errors are handled in the same manner; see lines 390 through 410.)

CAUSE ERROR and Error Numbers 1001 thru 1080

Error numbers 1001 through 1080 have been reserved to have special meaning for BASIC programs. These errors are used to simulate errors which may occur

when a binary has not been loaded. The value returned by ERRN will be 1; the ERRM\$ function will return either:

```
ERROR 1 in line_number Missing binary binary_number
```

or

```
ERROR 1 in line_number Missing binary binary_name
```

12

The second message is returned with language-extension binaries (no binary name is returned with driver binaries).

Clearing Error Conditions (CLEAR ERROR)

After you have finished handling an error in a program, it is convenient to clear the indications that an error has occurred. The following statement performs this action:

```
100 CLEAR ERROR
```

This statement has the following effects on the error functions' values:

ERRN	Subsequently returns 0.
ERRLN	Subsequently returns 0.
ERRL	Subsequently returns 0 for all arguments (line identifiers) sent to it.
ERRM\$	Subsequently returns the null string (string with length 0).
ERRDS	Is <i>not</i> affected by CLEAR ERROR.

Note that the CLEAR ERROR statement is *not* keyboard executable; it can only be executed from a running program.

Error handling for File Name Expansion

Certain file system commands can perform operations on multiple files when wildcards are enabled. Consult your *HP BASIC Language Reference* manual for a complete list of these commands.

All files that match a given wildcard argument will be processed by the command. For example, say you MSI to an SRM directory having the following CAT listing:

12

```
USERS/CHARLIE:REMOTE 21, 0
LABEL: Disc1
FORMAT: SDF
AVAILABLE SPACE:      60168

      SYS FILE  NUMBER  RECORD   MODIFIED   PUB  OPEN
FILE NAME      LEV TYPE  TYPE  RECORDS  LENGTH DATE     TIME  ACC  STAT
=====
AGENDA          1      ASCII    254      1 24-Apr-90 12:01 RW
MEMOS           1      DIR       12      24 22-Mar-89 23:12 RW
DATA1           1  98X6  BDAT      8      256  1-Apr-90 14:12 RW
DATA2           1  98X6  BDAT      7      256  2-Apr-90 14:12 RW
DATA3           1  98X6  BDAT      8      256  3-Apr-90 14:12 RW
DATA4           1  98X6  BDAT      5      256  4-Apr-90 14:12 RW
DATA5           1  98X6  BDAT      4      256  5-Apr-90 14:12 RW
DATA6           1  98X6  BDAT      8      256  6-Apr-90 14:12 RW
DATA7           1  98X6  BDAT      6      256  7-Apr-90 14:12 RW
DATA8           1  98X6  BDAT      5      256  8-Apr-90 14:12 RW
```

If HP-UX style wildcards were enabled previously, then you could copy all of the files with names prefixed by DATA to an existing directory named /USERS/FRED by entering the command:

```
COPY "DATA*" TO "/USERS/FRED"
```

If an exception occurs while processing one of the files, the name of the file will be displayed followed by a warning message. The command will then continue processing any remaining files which match the wildcard argument. After the command is finished processing the files, **ERROR 293 Operation failed on some files** will be generated.

Going back to the previous example, say that the `/USERS/FRED` directory already contained a file named `DATA1` before you executed the `COPY` command. If you typed:

```
COPY "DATA*" TO "/USERS/FRED"
```

the following warning message would be displayed to the IDISP line:

```
DATA1:ERROR 54 Duplicate file name.
```

The remaining files which matched the wildcard argument would be copied to the `/USERS/FRED` directory, and at end of the operation the following error would be generated:

```
ERROR 293 Operation failed on some files
```

Note that only one error is generated for a command no matter how many warning messages are generated for that command. The warning messages and error messages are all displayed on the IDISP line. Each message writes over the previous message.

To avoid missing any warning messages, you can enable `PRINTALL` by pressing the `PRINTALL` function key. By doing this, all warning messages and their corresponding error messages will go to the `PRINTALL IS` device. See the entry for `PRINTALL IS` in the *HP BASIC Language Reference*.

Recording Interaction and Troubleshooting Messages

Keeping track of computer interactions and error messages helps you to troubleshoot a program if something goes wrong. The `PRINTALL IS` statement allows you to do this. Information on this statement can be found in the following sections and chapters of this manual:

- If you are using `BASIC/WS` or `BASIC/DOS`, see the section called “Tracing” in the “Debugging Programs” chapter.
- If you are using `BASIC/UX`, see the section called “Using an HP-UX Printer Spooler” in the chapter called “Using a Printer.”

Debugging Programs

The problem of debugging a program is distinct from the issues raised in the “Handling Errors” chapter. The “Handling Errors” chapter is based on the premise that the programmer is 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 chapter 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, and all programmers should strive constantly to achieve this goal. Hopefully, the techniques that have 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.

Aside from recommended methods of writing software, the computer itself has several features which aid in the process of debugging.

Using Live Keyboard

One of the pleasing characteristics of your computer is that its keyboard is “live” even 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 instance, 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.)

13

Executing Commands While a Program Is Running

By way of illustration, key in the following program, press **RUN** (**f3** in the System, User 1, and User 2 menus of an ITF keyboard), and then execute the commands shown underneath the listing.

```
10  FOR I=1 TO 1.E+6
20  NEXT I
30  END

CAT
2+2
SQR(6^2+17.2^2)
PRINT "THE QUICK BROWN FOX"
TIMEDATE
```

Using Program Variables

Now, this program will take a fair amount of time to complete (on the order of minutes), so to find out how far the program has gone, look at the value of the variable I. Type:

I **Return** or **ENTER**.

The current value of I will be displayed at the bottom of the screen.

13-2 Debugging Programs

If you don't want to wait for the program to go through all one million iterations, you can merely change the value of I by entering:

```
I=999000
```

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 to be aware of is that the computer will only recognize variables that exist in the current program environment. For instance, 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
```

13

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, in the following program, it is perfectly legal to perform the operation ASSIGN @Dvm TO * from the keyboard, although it is not legal to perform ASSIGN @File TO "DATA" from the keyboard.

```
1  ASSIGN @Dvm TO 724
10 FOR I=1 TO 1.E+5
20 NEXT I
30 END
```

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

Calling Subprograms

Although the GOTO and GOSUB commands are illegal from the keyboard, it is perfectly legal to call subprograms from the keyboard. The parameters that are passed must either be constants or must be variables that exist in the current context. Also, the program in memory must be able to pass pre-run without errors.

Here is an example:

13

```
10 FOR I=1 TO 1E5
20 NEXT I
30 END
40 SUB Gather(INTEGER X)
50 OPTION BASE 1
60 DIM A(32)
70 CREATE BDAT "File"&VAL$(X),1
80 ASSIGN @Dvm TO 724
90 ASSIGN @File TO "File"&VAL$(X)
100 OUTPUT @Dvm;"N100S"
110 ENTER @Dvm;A(*)
120 OUTPUT @File;A(*)
130 PRINT A(*),
140 SUBEND
150 DEF FNPoly(X)
160 RETURN X^3+3*X^2+3*X+X
170 FNEEND
```

By executing `CALL Gather(1)` from the keyboard, the main program will be suspended while the subprogram is called, at which time a 1 record file will be opened, 32 readings will be taken from the voltmeter and stored in the file, and the readings will be printed on the screen. Then main program execution will resume where it left off.

Similarly, by executing `FNPoly(1)`, the value of the polynomial will be computed for $X=1$ and the answer (8) will be displayed at the bottom of the screen.

13-4 Debugging Programs

Pausing and Continuing a Program

You can also pause a program from the keyboard using the **PAUSE** (**Break**) key.

You may subsequently continue program execution:

- Press **CONTINUE** (**f2**) in the System menu of an ITF keyboard)
- Execute a CONT statement:

CONT **Return** or **ENTER**

or

CONT 100 **Return** or **ENTER**

or

CONT Line_label **Return** or **ENTER**

Note that a program which has been edited *cannot* be continued.

13

Keyboard Commands Disallowed During Program Execution

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:

CHANGE	FIND	SCRATCH
CONT	GET	SCRATCH A
COPYLINES	LOAD	SCRATCH BIN
DEL	MOVELINES	SCRATCH C
EDIT	RUN	SYSBOOT

Cross References

When debugging a program, and you think that the problem may be that you misspelled a variable name, you can use the XREF command to alphabetically list all variable names. This listing will also contain the line numbers where the variables were used, to help you locate any problems caused by misspelling or using the wrong variable.

13

Another way of using a cross-reference listing is when you need to find every place a particular variable name is used, but the system (and therefore the FIND command) is not available. It is often advisable to generate a cross reference at the end of a hard-copy (printer) listing of a large program. This information makes finding every occurrence of a variable much easier.

Generating a Cross-Reference Listing

The following XREF command prints a cross-reference listing on the default PRINTER IS device:

```
XREF
```

The next command sends a cross-reference to device selector 701:

```
XREF #701
```

Example Program and Cross Reference

Here is an example program, with a corresponding cross reference.

```
10 ! Fil "DoKeyFile"
20 DIM Key_value$(160)
30 INTEGER Key_number
40 CREATE BDAT "SOFTKEYS",3
50 ASSIGN @Keys TO "SOFTKEYS"
60 FOR I=0 TO 9
70   READ Key_number,Key_value$
80   OUTPUT @Keys;Key_number,Key_value$
90 NEXT I
100 ASSIGN @Keys TO *
110 LOAD KEY "SOFTKEYS"
120 ! ---- Key Data -----
130 DATA 8,"work!",5,"that",1,"See?",4,"you"
140 DATA 2,"I",3,"told",7,"would",6,"this"
150 END
```

13-6 Debugging Programs

Now generate a cross reference of the identifiers in the program:

XREF **Return** or **ENTER**

The following results are generated:

```
      >>>>  Cross Reference  <<<<

*  Numeric Variables
I          60   90
Key_number 30 <-DEF 70   80

*  String Variables
Key_value$ 20 <-DEF 70   80

*  I/O Path Names
@Keys      50   80  100

Unused entries = 7
```

13

This is not an exhaustive list of XREF outputs, since there were no COM blocks, subprogram calls, line labels, etc. However, it does give an idea of the general format of a cross-reference listing. (For a complete description of XREF listings, see the *HP BASIC Language Reference*.)

Note the <- DEF which appears in some of the line-number lists; this symbol appears when:

- The identifier is a variable in a formal parameter list (that is, in a SUB or DEF FN statement).
- The identifier is a variable declared in a COM, DIM, REAL, INTEGER, or COMPLEX statement.
- The identifier is a line label for that line.

Unused Entries

At the end of each context, a line is printed that begins with:

Unused entries =

The number of “unused entries” deals with the internal workings of the system. It tells how many symbol table entries are available:

- for which space has already been made
- but which are not currently used by a variable

This is a count of the symbol table entries which have been marked by pre-run as **unused**. Unreferenced symbol table locations which have not yet been marked unused by pre-run processing will show up in the lists of identifiers with empty reference lists. Note that a distinction is made here between **unused** and **unreferenced**.

13

Pre-run will convert **unreferenced** symbol table entries (entries which are *defined* by the system but not *used* by a variable in the program) into “unused” entries. Unreferenced entries can arise because you changed your mind about a variable’s name or corrected a typing error (once the system reserves space for a symbol table entry, this space is dedicated to the purpose of storing symbols until the corresponding context is destroyed, such as with SCRATCH). “Unreferenced entries” can also arise in syntaxing some statements where a numeric variable name which becomes a line label or a subprogram name is created. Also, REN (renumber) can cause line numbers to merge if you have unsatisfied line-number references. This shows up in the cross-reference as separate (but adjacent) entries for the multiple symbol table entries for the line number.

Let’s go through an example to make this completely clear. At power-up the system creates an empty symbol table with space for five entries. Doing an XREF at this point will show **Unused entries = 5** and no other symbols.

Now type in the following program:

```
10 A=1
20 B=2
30 C=3
40 D=4
50 E=5
```

An XREF at this point will show five variables, each occurring in one line, and **Unused entries = 0** (all the pre-allocated spaces having been filled).

Now add one more line:

```
60 F=6
```

13-8 Debugging Programs

A subsequent XREF will show six variables and **Unused entries = 5**. This happens because when the system needs a symbol table location and none exists it always allocates six additional spaces: one for its immediate needs, and five spares for future use.

Now delete lines 10 to 60 using DEL 10,60 or the **Delete line** (**DEL LN**) key. Then perform another XREF. The listing will show six variables, each with an empty reference list, and **Unused entries = 5**.

Now store the following program line (as the only line):

```
10 END
```

and run the program. Now an XREF will show **Unused entries = 11**.

Doing a SCRATCH will restore the initial state with the symbol table reduced to five empty locations.

Now enter the following program lines:

```
10 GOTO A
20 A:END
```

Then execute XREF. This will show a numeric variable A (which is an artifact of the syntaxing process) and the line label A (referenced in two places). Running this program will cause pre-run to recognize that there is no occurrence of a numeric variable A in the program and reclaim the space for future use, converting it back into an “unused entry”. Variables which are defined in the program are considered “referenced” and cannot be converted to “unused” even if no assignment or access is made to them, because they must be present in the symbol table in order for the program to list. Such variables must be found by looking at the XREF for variables with reference lists which contain only defining occurrences (<-DEF).

Single-Stepping a Program

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 the variables and the sequence in which the program is running at each statement. This is done with the **STEP** key (**f1** in the System menu of an ITF keyboard).

There are three ways to use the **STEP** key:

1. If the program is stopped (i.e., a pre-run has to be performed), pressing the **STEP** key 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 the **STEP** key again will cause that line to be executed, and the next line after that to be executed will appear in the message line. If the **STEP** key 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 the **STEP** key 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.
2. If the program is in an INPUT or LINPUT statement, pressing the **STEP** key is sufficient to terminate the operation. Any data entered from the keyboard will be entered into the correct variables, just as though **CONTINUE** (**f2** on the ITF keyboard) or **ENTER** (**Return** on the ITF keyboard) 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 the **STEP** key will cause the next line to be executed. The program counter will not be reset, nor will a pre-run 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 the **STEP** key repeatedly.

13-10 Debugging Programs

```
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 the **STEP** key 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.

If you are stepping a program and encounter an INPUT, LINPUT, or ENTER KBD statement, you can use **Return**, **ENTER**, or **CONTINUE** to enter your responses. The system will remember that you are stepping the program and remain in single-step mode after the input operation is complete (unless you press **CONTINUE** again *after* the input operation is complete).

If you hold down the **STEP** key, to continuously step through program lines, you may want to turn softkey labels off (especially when using bit-mapped alpha displays).



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 with the **PRT ALL** key (**f4** in the System menu of an ITF keyboard), unless of course you're a very fast reader. (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.)

Turn Print All ON and key in the following example to see how TRACE ALL works:

13

```
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. The following example illustrates the use of one optional line specifier:

```
1 TRACE ALL 40
10 DIM A(1:10)
20 FOR I=1 TO 100
30 NEXT I
40 FOR J=1 TO 10
50 A(J)=J
60 NEXT J
70 END
```

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. In the above example, to trace the loop from 20 to 30 instead of the one from 40 to 60, simply delete line 1 and type in TRACE 20,40 from the keyboard.

```

10 DIM A(1:10)
20 FOR I=1 TO 100
30 NEXT I
40 FOR J=1 TO 10
50 A(J)=J
60 NEXT J
70 END

```

The program will begin tracing at line 20, 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. Thus in the following program, even though the TRACE ALL Printout statement is executed inside the subprogram, tracing does not commence until the subprogram has been exited and the Printout statement in the main program has been executed.

```

10 DIM A(1:10)
20 FOR I=1 TO 10
30     CALL Dummy(A(*),I)
40     GOSUB Printout
50 NEXT I
60 STOP
70 Printout: !
80 FOR J=1 TO 10
90 PRINT A(J);", ";
100 NEXT J
105 PRINT
110 RETURN
120 END
130 SUB Dummy(X(*),Z)
140 TRACE ALL Printout
150 FOR I=1 TO 10
160     X(I)=Z*100+I
170 NEXT I

```

```
180 GOSUB Printout
190 SUBEXIT
200 Printout: !
210 PRINT "Dummy routine executed";Z
220 RETURN
230 SUBEND
```

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.

13

PRINTALL IS

The PRINTALL IS command is useful for switching the tracing messages between the CRT and a hardcopy printer. For instance, turning PRINTALL ON during pre-run will allow you to see which array variable has not been dimensioned. (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 (PRINTALL IS 701).

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 with the **CONTINUE** key (**f2**) in the System and User menus on an ITF keyboard), the **STEP** key (**f1**) in the System menu on an ITF keyboard), or by executing CONT 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 will pause every time it reaches line 70.

```

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

```

Try the following ways of continuing execution:

- press **STEP** (**f1**) on the ITF keyboard)
- press **CONTINUE** (**f2**) on the ITF keyboard)
- execute **CONT 110**

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 (Break) Key

The **CLR I/O** key (**Break**) on the ITF keyboard) suspends any active I/O operation and pauses the program in such a way that the suspended statement will restart once **CONTINUE** (**f2**) on the ITF keyboard) or **STEP** (**f1**) on the ITF keyboard) 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, **CLR I/O** can help in debugging these situations.

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

13

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

Introduction to I/O

This chapter is an introduction to I/O (Input/Output) programming concepts and programming techniques. The remainder of this manual (chapters 15 through 21) covers specific interfacing topics.

Interfacing Concepts

This section introduces some interfacing concepts important to I/O programming. Let's begin by defining some terms.

14

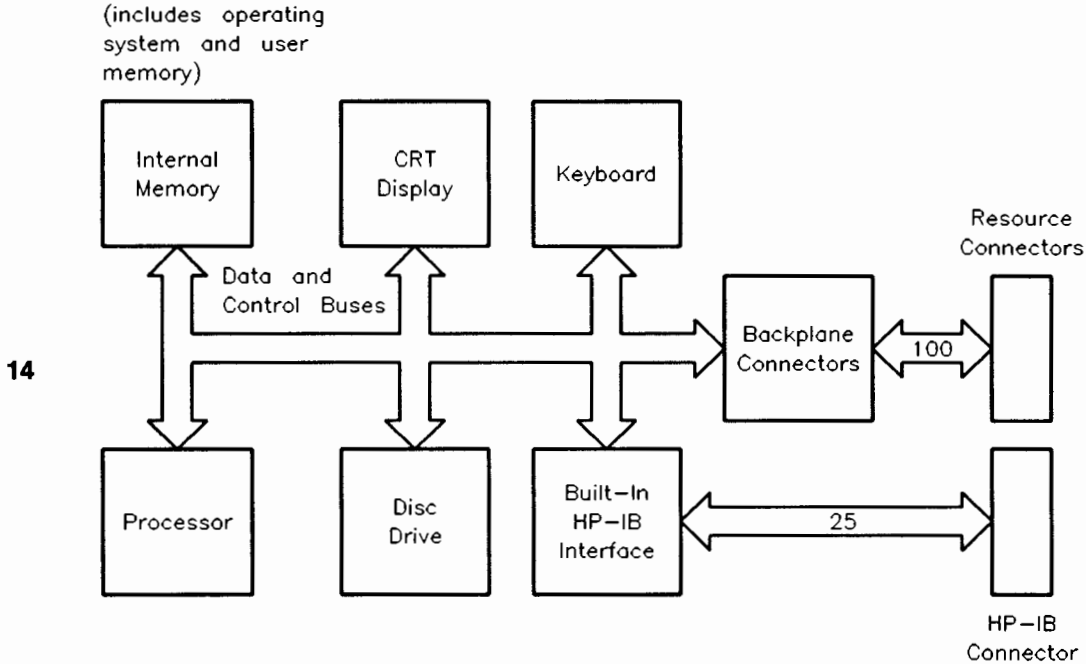
Terminology

These terms are important to your understanding of the text of the remainder of this manual. The purpose of this section is to make sure that our terms have the same meanings.

computer	is defined for our purposes to be the processor, its support hardware, and the BASIC-language operating system; together these system elements <i>manage</i> all computer resources.
hardware	describes both the electrical connections and electronic devices that make up the circuits within the computer; any piece of hardware is an actual physical device.
software	describes the user-written, BASIC-language programs.
firmware	refers to the pre-programmed, machine-language programs that are invoked by BASIC-language statements and commands. As the term implies, firmware is not usually modified by BASIC users. The machine-language routines of the operating system are firmware programs.

computer resource

is used in this manual to describe all of the “data-handling” elements of the system. Computer resources include: internal memory, CRT display, keyboard, and disk drive, and any external devices that are under computer control.



Block Diagram of the Computer

I/O

is an acronym that comes from “Input and Output”; it refers to the process of copying data to or from computer memory.

output

involves moving data from computer memory to another resource. During output, the **source** of data is computer memory and the **destination** is any resource, including memory.

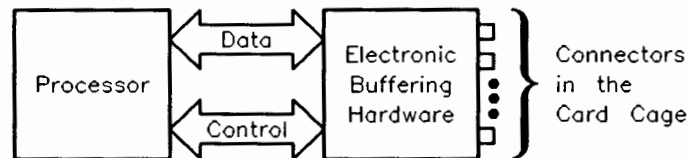
input

is moving data from a resource to computer memory; the source is any resource and the destination is a variable in computer memory. *Inputting data is also referred to as “entering data” in this manual for the sake of avoiding confusion with the INPUT statement.*

14-2 Introduction to I/O

bus refers to a common group of hardware lines that are used to transmit information between computer resources. The computer communicates directly with the internal resources through the data and control buses.

computer backplane is an extension of these internal data and control buses. The computer communicates indirectly with the external devices through interfaces connected to the backplane hardware.



Backplane Hardware

Why Do You Need an Interface?

14

The primary function of an interface is, obviously, to provide a communication path for data and commands between the computer and its resources. Interfaces act as intermediaries between resources by handling part of the “bookkeeping” work, ensuring that this communication process flows smoothly. The following paragraphs explain the need for interfaces.

First, even though the computer backplane is driven by electronic hardware that generates and receives electrical signals, this hardware was not designed to be connected directly to external devices. The electronic backplane hardware has been designed with specific electrical logic levels and drive capability in mind.

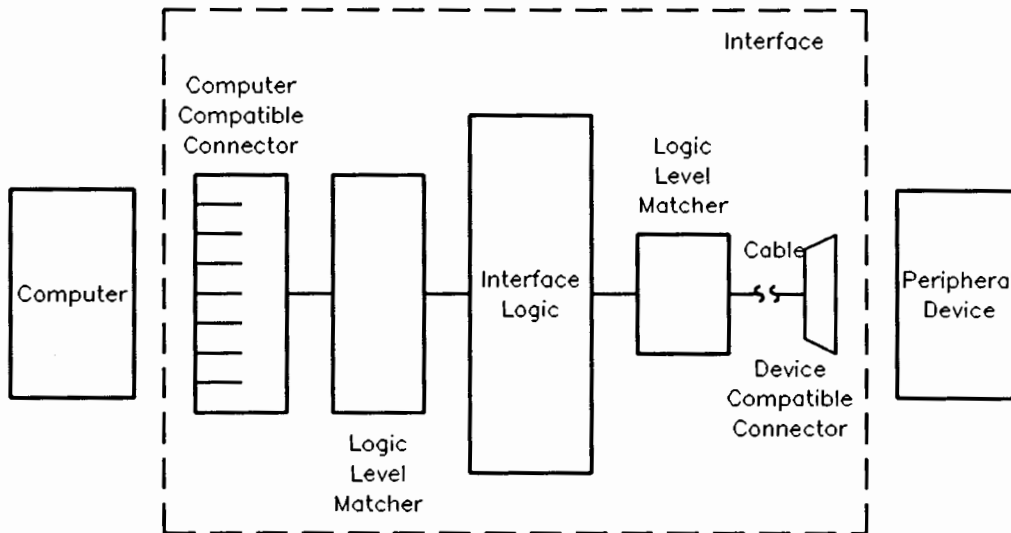
Caution Exceeding backplane hardware ratings will damage this electronic hardware.



Second, you cannot be assured that the connectors of the computer and peripheral are compatible. In fact, there is a good probability that the connectors may not even mate properly, let alone that there is a one-to-one correspondence between each signal wire’s function.

Third, assuming that the connectors and signals are compatible, you have no guarantee that the data sent will be interpreted properly by the receiving device. Some peripherals expect single-bit serial data while others expect data to be in 8-bit parallel form.

Fourth, there is no reason to believe that the computer and peripheral will be in agreement as to when the data transfer will occur; and when the transfer does begin the transfer rates will probably not match. As you can see, interfaces have a great responsibility to oversee the communication between computer and its resources. The functions of an interface are shown in the following block diagram.



Functional Diagram of an Interface

Electrical and Mechanical Compatibility

Electrical compatibility must be ensured before any thought of connecting two devices occurs. Often the two devices have input and output signals that do not match; if so, the interface serves to match the electrical levels of these signals before the physical connections are made.

Mechanical compatibility simply means that the connector plugs must fit together properly. All of the HP Series 200/300 interfaces have 100-pin connectors that mate with the computer backplane. The peripheral end of the

14-4 Introduction to I/O

interfaces may have unique configurations due to the fact that several types of peripherals are available that can be operated with the computer. Most of the interfaces have cables available that can be connected directly to the device so you don't have to wire the connector yourself.

Data Compatibility

Just as two people must speak a common language, the computer and peripheral must agree upon the form and meaning of data before communicating it. As a programmer, one of the most difficult compatibility requirements to fulfill before exchanging data is that the format and meaning of the data being sent is identical to that anticipated by the receiving device. Even though some interfaces format data, most interfaces have little responsibility for matching data formats; most interfaces merely move agreed-upon quantities of data to or from computer memory. The computer must generally make the necessary changes, if any, so that the receiving device gets meaningful information.

14

Timing Compatibility

Since all devices do not have standard data-transfer rates, nor do they always agree as to when the transfer will take place, a consensus between sending and receiving device must be made. If the sender and receiver can agree on both the transfer rate and beginning point (in time), the process can be made readily.

If the data transfer is not begun at an agreed-upon point in time and at a known rate, the transfer must proceed one data item at a time with acknowledgement from the receiving device that it has the data and that the sender can transfer the next data item; this process is known as a "handshake". Both types of transfers are utilized with different interfaces and both will be fully described as necessary.

Additional Interface Functions

Another powerful feature of some 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. The

actual tasks performed by each type of interface card vary widely. Each interface is described in detail in Volume 2 of this manual.

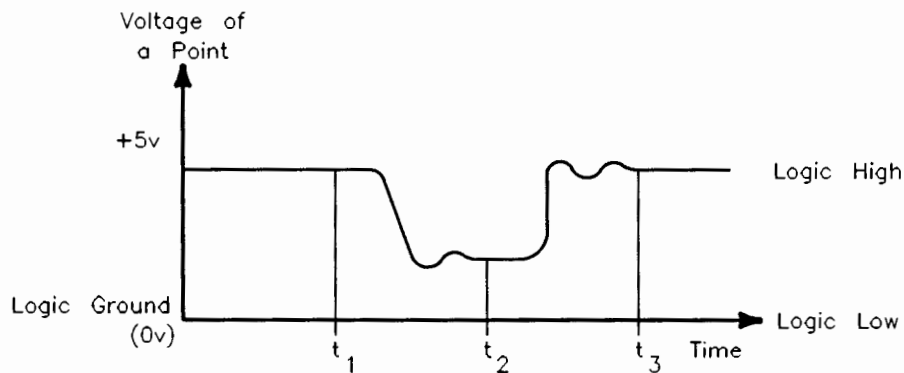
Data Representations

As long as data is only being used internally, it really makes little difference how it is represented; the computer always understands its own representations. However, when data is to be moved to or from an external resource, the data representation is of paramount importance.

Bits and Bytes

Computer memory is no more than a large collection of individual bits (*binary digits*), each of which can take on one of two logic levels (high or low). Depending on how the computer interprets these bits, they may mean on or not on (off), true or not true (false), one or zero, busy or not busy, or any other bi-state condition. These logic levels are actually voltage levels of hardware locations within the computer. The following diagram shows the voltage of a point versus time and relates the voltage levels to logic levels.

14



Voltage and Positive-True Logic

In some cases, you want to determine the state of an individual bit (of a variable in computer memory, for instance). The logical binary functions (BIT, BINCOMP, BINIOR, BINEOR, BINAND, ROTATE, and SHIFT) provide access to the individual bits of data.

14-6 Introduction to I/O

In most cases, these individual bits are not very useful by themselves, so the computer groups them into multiple-bit entities for the purpose of representing more complex data. Thus, all data in computer memory are somehow represented with binary numbers.

The computer's hardware accesses groups of sixteen bits at one time through the internal data bus; this size group is known as a **word**. With this size of bit group, 65 536 ($65\,536=2^{16}$) different bit patterns can be produced. The computer can also use groups of eight bits at a time; this size group is known as a **byte**. With this smaller size of bit group, 256 ($256=2^8$) different patterns can be produced. How the computer and its resources interpret these combinations of ones and zeros is very important and gives the computer all of its utility.

Representing Numbers

The following binary weighting scheme is often used to represent numbers with a single data byte. Only the non-negative integers 0 through 255 can be represented with this particular scheme.

← Most-Significant Bit				Least-Significant Bit →			
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	1	0	1	1	0
value=128	value=64	value=32	value=16	value=8	value=4	value=2	value=1

Notice that the value of a 1 in each bit position is equal to the power of two of that position. For example, a 1 in the 0th bit position has a value of 1 ($1=2^0$), a 1 in the 1st position has a value of 2 ($2=2^1$), and so forth. The number that the byte represents is then the total of all the individual bit's values.

$$0 \times 2^0 = 0$$

$$1 \times 2^1 = 2$$

$$1 \times 2^2 = 4$$

$$0 \times 2^3 = 0$$

$$1 \times 2^4 = 16$$

$$0 \times 2^5 = 0$$

$$0 \times 2^6 = 0$$

$$1 \times 2^7 = 128$$

$$\text{Number represented} = 2 + 4 + 16 + 128 = 150$$



The preceding representation is used by the "NUM" function when it interprets a byte of data. The next section explains why the character "A" can be represented by a single byte.

```
100 Number=NUM("A")
110 PRINT " Number = ";Number
120 END
```

Number = 65 *Printed Result*

Representing Characters

Data stored for humans is often alphanumeric-type data. Since less than 256 characters are commonly used for general communication, a single data byte can be used to represent a character. The most widely used character set is defined by the ASCII standard. (ASCII stands for "American Standard Code for Information Interchange". See the Useful Tables section in volume 2 of the *HP BASIC Language Reference* for the complete table.)

14

This standard defines the correspondence between characters and bit patterns of individual bytes. Since this standard only defines 128 patterns (bit 7 = 0), 128 additional characters are defined by the computer (bit 7 = 1). The entire set of the 256 characters on the computer is hereafter called the "extended ASCII" character set.

When the CHR\$ function is used to interpret a byte of data, its argument must be specified by its binary-weighted value. The single (extended ASCII) character returned corresponds to the bit pattern of the function's argument.

```
100 Number=65                    ! Bit pattern is "01000001"
110 PRINT " Character is ";
120 PRINT CHR$(Number)
130 END
```

Character is A *Printed Result*

Representing Signed Integers

There are two ways that the computer represents signed integers. The first uses a binary weighting scheme similar to that used by the NUM function. The second uses ASCII characters to represent the integer in its decimal form.

14-8 Introduction to I/O

Internal Representation of Integers. Bits of computer memory are also used to represent signed (positive and negative) integers. Since the range allowed by eight bits is only 256 integers, a word (two bytes) is used to represent integers. With this size of bit group, 65536 ($65536=2^{16}$) unique integers can be represented.

The range of integers that can be represented by 16 bits can arbitrarily begin at any point on the number line. In the computer, this range of integers has been chosen for maximum utility; it has been divided as symmetrically as possible about zero, with one of the bits used to indicate the sign of the integer.

With this "2's-complement" notation, the most significant bit (bit 15) is used as a sign bit. A sign bit of 0 indicates positive numbers and a sign bit of 1 indicates negatives. You still have the full range of numbers to work with, but the range of absolute magnitudes is divided in half (-32768 through 32767). The following 16-bit integers are represented using this 2's-complement format.

Binary Representation				Decimal Equivalent
1111	1111	1111	1111	-1
0000	0000	0000	0000	1
1111	1111	0000	0001	-255
0000	0000	1111	1111	255
↖	↑	↑	↑	
sign	2^{14}	2^{13}	2^8	2^7
bit				2^0

The representation of a positive integer is generated according to place value, just as when bytes are interpreted as numbers. To generate a negative number's representation, first derive the positive number's representation. Complement (change the ones to zeros and the zeros to ones) all bits, and then to this result add 1. The final result is the two's-complement representation of the negative integer. This notation is very convenient to use when performing math operations. Let's look at a simple addition of 2 two's-complement integers.

Example: 3 + (-3) = ?

First, +3 is represented as: 0000 0000 0000 0011

Now generate -3's representation:

first complement +3, 1111 1111 1111 1100
then add 1 + 0000 0000 0000 0001

-3's representation: 1111 1111 1111 1101

Now add the two numbers: 1111 1111 1111 1101
+ 0000 0000 0000 0011

*final carry
not used*

1← 1← *carry on
0000 0000 0000 0000 all places*

14

ASCII Representation of Integers. ASCII digits are often used to represent integers. In this representation scheme, the decimal (rather than binary) value of the integer is formed by using the ASCII digits 0 through 9 {CHR\$(48) through CHR\$(57), respectively}. An example is shown below.

Example

The decimal representation of the binary value "1000 0000" is 128. The ASCII-decimal representation consists of the following three characters.

Character	Decimal Code	Binary Code
1	49	00110001
2	50	00110010
8	56	00111000

Representing Real Numbers

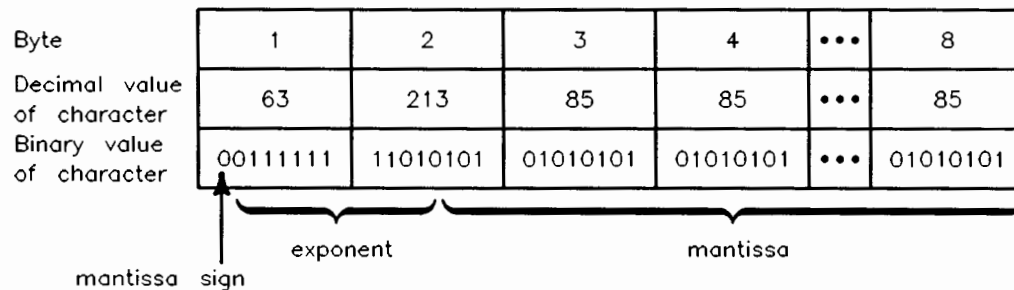
Real numbers, like signed integers, can be represented in one of two ways with the computers. They are represented in a special binary mantissa-exponent notation within the computers for numerical calculations. During output and enter operations, they can also be represented with ASCII-decimal digits.

Internal Representation of Real Numbers. Real numbers are represented internally by using a special binary notation. With this method, all numbers of the REAL data type are represented by eight bytes: 52 bits of mantissa magnitude, 1 bit for mantissa sign, and 11 bits of exponent. The following equation and diagram illustrate the notation; the number represented is 1/3.

Note



The internal representation used for real numbers is the IEEE standard 64-bit floating-point notation. For further details, refer to chapter 3, "Numeric Computation."



$$\text{Real number} = (-1)^{\text{mantissa sign}} * 2^{\text{exponent}-1023} * (1.\text{mantissa})$$

Even though this notation is an international standard, most external devices don't use it; most use an ASCII-digit format to represent decimal numbers. The computer provides a means so that both types of representations can be used during I/O operations.

ASCII Representation of Real Numbers. The ASCII representation of real numbers is very similar to the ASCII representation of integers. Sign, radix, and exponent information are included with ASCII-decimal digits to form these number representations. The following example shows the ASCII representation of 1/3. Even though, in this case, 18 characters are required to get the same accuracy as the eight-byte internal representation shown above, not all real numbers represented with this method require this many characters.

ASCII characters	0	.	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Decimal value of characters	48	46	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51

14

The I/O Process

When using statements that move data between memory and internal computer resources, you do not usually need to be concerned with the details of the operations. However, you may have wondered how the computer moves the data. This section takes you “behind the scenes” of I/O operations to give you a better intuitive feel for how the computer outputs and enters data.

I/O Statements and Parameters

The I/O process begins when an I/O statement is encountered in a program. The computer first determines the type of I/O statement to be executed (such as, OUTPUT, ENTER USING, etc.). Once the type of statement is determined, the computer evaluates the statement’s parameters.

Specifying a Resource. Each resource must have a unique specifier that allows it to be accessed to the exclusion of all other resources connected to the computer. The methods of uniquely specifying resources (output destinations and enter sources) are device selectors, string variable names, and I/O path names. These specifiers are further described in the section “Directing Data Flow,” later in this chapter.

For instance, before executing an OUTPUT statement, the computer first evaluates the parameter which specifies the destination resource. The source parameter of an ENTER statement is evaluated similarly.

```
OUTPUT Dest_parameter;Source_item
```

```
ENTER Sourc_parameter;Dest_item
```

Firmware. After the computer has determined the resource with which it is to communicate, it “sets up” the moving process. The computer chooses the method of moving the specified data according to the type of resource specified and the type of I/O statement. The actual machine-language routine that executes the moving procedure is in firmware. Since there are differences in how each resource represents and transfers data, a dedicated firmware routine must be used for each type of resource. After the appropriate firmware routine has been selected, the next parameter(s) must be evaluated (i.e., source items for OUTPUT statements and destination items for ENTER statements).

Registers. The computer must often read certain memory locations to determine which firmware routines will be called to execute the I/O procedure. The content of these locations, known as registers, store parameters such as the type of data representation to be used and type of interface involved in the I/O operation.

An example of register usage by firmware is 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 the 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.

The program can also determine the contents of these registers. The statements that provide access to the registers are described in chapter 17. The contents of all registers accessible by the program are described in the *HP BASIC 6.2 Interface Reference* manual.

Data Handshake

Each byte (or word) of data is transferred with a procedure known as a data-transfer handshake (or simply “handshake”). It is the means of moving one byte of data at a time when the two devices are not in agreement as to the rate of data transfer or as to what point in time the transfer will begin. The steps of the handshake are as follows.

1. The sender signals to get the receiver’s attention.
2. The receiver acknowledges that it is ready.
3. A data byte (or word) is placed on the data bus.
4. The receiver acknowledges that it has gotten the data item and is now busy. No further data may be sent until the receiver is ready.
5. Repeat these steps if more data items are to be moved.

14

I/O Examples

Now that you have seen all of the steps taken by the computer when executing an I/O statement, let’s look at how two typical I/O statements are executed by the computer.

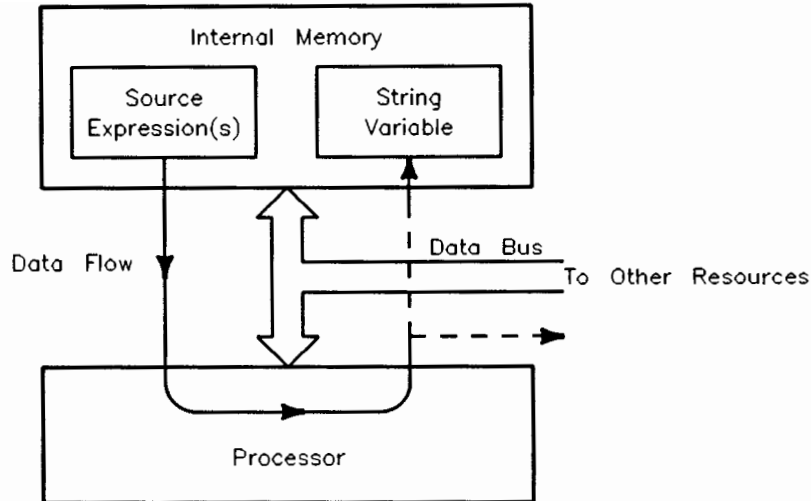
Example Output Statement

Data can be output to only one resource at a time with the OUTPUT statement (with the exception of the HP-IB Interface). This destination can be any computer resource, which is specified by the destination parameter as shown below.

```

      Destination parameter
      /
OUTPUT Destination;String$,CHR$(C+32),"That's all"
      |   Source items are expressions   |
```

The source of data for output operations is always memory. Either string or numeric expressions can specify the actual data to be output. The flow of data during output operations is shown below. Notice that all data copied from memory to the destination resource by the OUTPUT statement passes through the processor under the control of operating-system firmware.



Data Flow During Output Operations

14

Source-Item Evaluation. The source items, listed after the semicolon and separated by commas, can be any valid numeric or string expression. As the statement is being executed, these expressions must be individually evaluated and the resultant data representation sent to the specified destination. The results of the evaluation depend on the type of expression (numeric or string) and on which data representation (ASCII or internal) is to be used during the I/O operation.

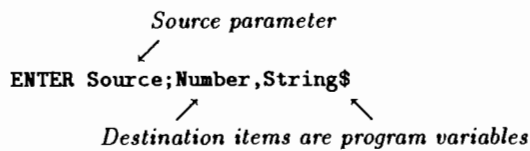
If the expression is a variable *and* the internal data representation is to be used, the data is ready to be copied byte-serially (or word-serially) to the destination; otherwise, the expression must be completely evaluated. The representation generated during the evaluation is stored in a temporary variable within memory. In both cases, once the beginning memory location and length of the data are known, the copying process can be initiated.

Copying Data to the Destination. The computer employs “memory-mapped” I/O operations; all devices are addressable as memory locations. All output operations involve a series of two-step processes. The first step is to copy one byte (or word) from memory into the processor. The second step is then to copy this byte (or word) into the destination location (a memory address). Each item in the list is output in this serial fashion. The appropriate handshake firmware routine is executed for each byte (or word) to be copied.

Since there may be several data items in the source list, it may be necessary to output an item-terminator character after each item to communicate the end of the item to the receiver. If the item is the last item in the source list, the computer may signal the receiver that the output operation is complete. Either an item terminator or end-of-line sequence of characters can be sent to the receiver to signal the end of this data transmission. The OUTPUT statement is described in full detail in chapter 15.

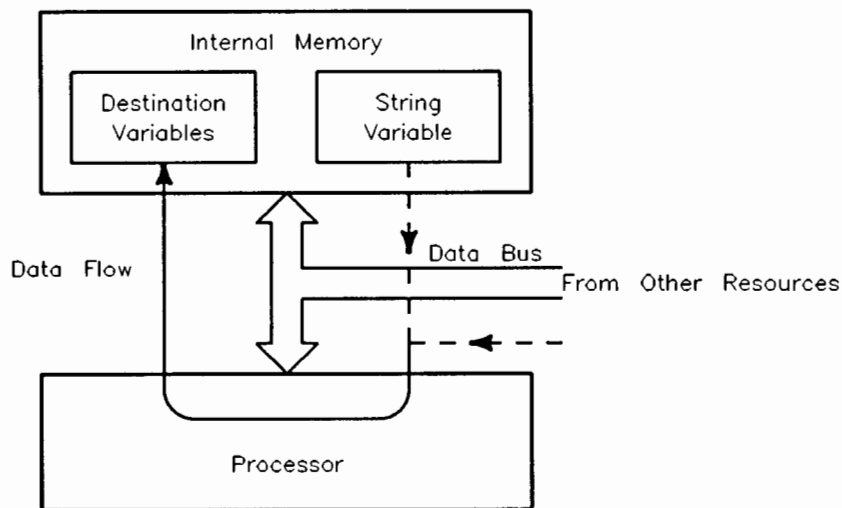
Example Enter Statement

Data can be entered from only one resource at a time. This source can be any resource and is specified by the source parameter as shown in the following statement.



14

The destinations of enter operations are always variables in memory. Both string and numeric variables can be specified as the destinations. The flow of data during enter operations is shown below.



Data Flow During Enter Operations

Destination-Item Evaluation. The destination(s) of data to be entered is (are) specified in the destination list. Either string or numeric variables can be specified, depending on the type of data to be entered. In general, as each destination item is evaluated, the computer finds its actual memory location so that data can be copied directly into the variable as the enter operation is executed. However, if the ASCII representation is in use, numeric data entered is stored in a temporary variable during entry.

Copying Data into the Destinations. As with output operations, entering data is a series of two-step processes. Each data byte (or word) received from the sender is entered into the processor by the appropriate handshake firmware. It is then copied into either a temporary variable or a program variable. If more than one variable is to receive data, each incoming data item must be properly terminated. If the internal representation is in use, the computer knows how many characters are to be entered for each variable. If the ASCII representation is in use, a terminator character (or signal) must be sent to locate the end of each data item. When all data for the item has been received, it is evaluated, and the resultant internal representation of the number is placed into the appropriate program variable. Further details concerning the ENTER statement are contained in chapter 16.

14

Sharing Resources in a Multi-Tasking/Multi-User Environment (BASIC/UX only)

HP-UX provides BASIC/UX with a multi-tasking/multi-user environment. This section explains the concepts of multi-tasking and multi-user environments and some of the things you need to be aware of in these environments. Topics covered are:

- What is multi-tasking?
- What does multi-user mean?
- Using I/O resources in a multi-tasking/multi-user environment

What Is Multi-Tasking?

Multi-tasking means you can perform more than one task at a time. These tasks are referred to as processes when running BASIC/UX on HP-UX. For example, you can run BASIC/UX in two windows at the same time and have a different program or process running in each window. This capability helps you to be more productive as a programmer.

What Does Multi-User Mean?

Multi-user systems allow more than one user to be on the computer at the same time. What this means is your computer can have several terminals connected to it with each terminal having access to all of the system resources.

Using I/O Resources in a Multi-Tasking/Multi-User Environment

14

With BASIC/UX running on HP-UX, you need to be alert to the fact that the shared resources such as disks, I/O interfaces, printers, plotters, etc. may be used by other people on the system. Therefore, it is a good idea to practice good “system citizenship” and not try to use a resource that someone else is currently using. In the case of printer and plotter spoolers, good citizenship is enforced.

As a BASIC/UX user on HP-UX you are responsible for the following:

- Interfaces should be locked only as long as needed (unused interfaces should be freed for use by other waiting processes).
- Do not lock any interface containing a swap device (e.g., the system disk).
- Do not lock any interfaces containing mounted (HFS) file systems.
- Since system RAM is shared among the kernel, BASIC/UX, and other processes, arbitrarily allocating huge workspaces will severely limit multi-tasking capabilities and may cause system performance to deteriorate to an unacceptable level. Similarly, locking processes into memory can result in severe performance degradation to competing processes and should be done sparingly, if at all.
- Refrain from unnecessarily setting a high real-time run priority.

14-18 Introduction to I/O

- Shared system resources, such as printer spoolers, should not be accessed directly. Simultaneous access by multiple processes can result in another person's output being mixed with yours (interleaved).

For further information about I/O in the HP-UX environment, refer to *HP BASIC 6.2 Advanced Programming Techniques*.

Directing Data Flow

As described previously in this chapter, data can be moved between computer memory and several resources, including:

- Computer memory (BASIC string variables).
- Internal devices (such as the display and keyboard).
- External devices (such as instruments and printers).
- HP-UX pipes (BASIC/UX only).
- Windows (BASIC/UX only).
- Mass storage files (on disk or in memory volumes).
- Buffers (variables in memory with special capabilities for high-speed, background-process transfers).

Here is where to learn about how to use each of these I/O resources:

I/O Resource	Where Covered
String variables, devices, pipes, and windows	are described in this chapter.
Files	are briefly described in the "I/O Path Attributes" chapter of this manual, and more fully in the "Data Storage and Retrieval" chapter.
Buffers	are described in the "Transfers and Buffered I/O" chapter of this manual.

Specifying a Resource

Each resource must have a specifier that allows it to be accessed to the exclusion of all other computer resources. String variables are specified by variable name, while devices can be specified by either their device selector or a data type known as an I/O path name. This section describes how to specify these resources in OUTPUT and ENTER statements.

String-Variable Names

Data is moved to and from string variables by specifying the string variable's name in an OUTPUT or ENTER statement. Examples of each are shown in the following program (lines 200 and 240).

14

```
100 DIM To_string$(80),From_string$(80)
110 DIM Data_out$(80)
120 !
130 From_string$="Source data"
140 Data_out$="OUTPUT data"
150 !
160 PRINTER IS CRT
170 PRINT "To_string$ before OUTPUT = ";To_string$
180 PRINT
190 !
200 OUTPUT To_string$;Data_out$; ! ";" suppresses CR/LF.
210 PRINT "To_string$ after OUTPUT = ";To_string$
220 PRINT
230 !
240 ENTER From_string$;To_string$
250 PRINT "To_string$ after ENTER = ";To_string$
260 PRINT
270 !
280 END
```

Printed Results

```
To_string$ before OUTPUT=(null string)

To_string$ after OUTPUT= OUTPUT data

To_string$ after ENTER= Source data
```

Data is always copied to the destination string (or from the source string) beginning at the first position of the variable; subscripts cannot be used to specify any other beginning position within the variable.

Formatted String I/O. The use of outputting to and entering from string variables is a very powerful method of buffering data to be output to other resources. With OUTPUT and ENTER statements that use images, the data sent to the string variables can be explicitly formatted before being sent to (or while being received from) the variable. Further uses of string variables are described in the "Concepts of Unified I/O" chapter in *HP BASIC 6.2 Advanced Programming Techniques*.

Device Selectors

Devices include the built-in CRT and keyboard, external printers and instruments, and all other physical entities that can be connected to the computer through an interface. Each interface has a unique number by which it is identified, its **interface select code**.

In order to send data to or receive data from a device, merely specify the select code of its interface in an OUTPUT or ENTER statement. Examples of using select codes to access devices are shown below.

```
OUTPUT 1;"Data to CRT"  
ENTER CRT;Crt_line$
```

The expressions 1 and CRT are device selectors for the CRT display.

```
Serial=9  
ENTER Serial;Serial_data$  
OUTPUT Serial;"To serial card"
```

9 is the device selector for a serial interface.

```
Hpib_device=722  
OUTPUT 722;"F1R1"  
ENTER Hpib_device;Reading
```

722 is the device selector of an HP-IB device.

The following pages explain select codes and device selectors for HP 9000 Series 200/300 computers. For information about select codes for the HP Measurement Coprocessor, refer to *Installing and Using HP BASIC/DOS 6.2*.

Note

When using devices with BASIC/UX from within a multi-user environment, you should either:

- Use a pipe to a spooler associated with a device.
- Lock the interface connected to a device before using it.

Select Codes of Built-In Interfaces (Series 200/300). The internal devices are accessed with the following, permanently assigned interface select codes.

Select Codes of Built-In Devices

Built-In Interface/Device	Permanent Select Code
Alpha Display	1
Keyboard	2
Graphics Display (<i>non-bit mapped alpha/graphics displays</i>) ¹	3
Flexible Disk Drive (Models 226 and 236 only) ¹	4
Graphics Display (<i>bit-mapped alpha/graphics displays</i>)	6 or 132
Built-in HP-IB interface	7
Built-in serial interface	9
SCSI (Models 345, 362, 375, 380, and 382)	14
Local Area Network (LAN)	21
HP Parallel (Models 345, 362, 375, 380, 382)	23
SCSI (Model 340)	28
Parity-checking (memory), cache, and floating-point math hardware	32 (pseudo)
EXT SIGNAL Registers (BASIC/UX)	33 (pseudo)

¹ Available for BASIC/WS only.

Select Codes of Optional Interfaces (Series 200/300). Optional interfaces all have switch-settable select codes. The valid range of select codes is 8 through 31 (they *cannot* use select codes 1 through 7, since these are reserved for built-in devices). 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. See the interface's installation manual for further instructions.

Factory Select-Code Settings for Optional Interfaces

Optional Interface	Default Select Code
HP-IB (HP 98624)	8
Serial (HP 98626, HP 98644)	9 ¹
BCD (HP 98623) ²	11
GPIO (HP 98622)	12
High-Speed (HP-IB) Disk (HP 98625)	14
SCSI (HP 98658A)	14
4-channel Multiplexer (HP 98642) ³	16
Data Communications (HP 98628)	20
Local Area Network (LAN) (HP 98643) ¹	21
Shared Resource Manager (HP 98629)	21
EPROM Programmer (HP 98253) ²	27
Color Output (HP 98627)	28
Bubble Memory (HP 98259) ²	30

14

¹ Use another select code if there is already a built-in serial interface at this select code.

² Available for BASIC/WS only.

³ Available for BASIC/UX only.

HP-IB Device Selectors. Each device on the HP-IB interface has a **primary address** by which it is uniquely identified; each address must be unique so that only one device is accessed when one address is specified. The device selector is then a combination of the interface select code and the device's address. Some examples are shown below. (The HP-IB also has additional capabilities that add to this definition of device selectors. See the chapter called "The HP-IB Interface" for further details.)

HP-IB Device Selector Examples

Device Location	Device Selector	Example I/O Statement
interface select code 7, primary address 22	722	OUTPUT 722;"Data" ENTER 722;Number
interface select code 10, primary address 13	1013	OUTPUT 1013;"Data" ENTER 1013;Number
interface select code 10, primary address 01	1001	OUTPUT 1013;"Data" ENTER 1013;Number

Multiplexer Device Selectors (BASIC/UX Only). Each device on an HP 98642 Multiplexer interface has a **channel number** by which it is uniquely identified. (This channel number is similar to the primary address of the HP-IB device selectors shown above.) Some examples are shown below.

Multiplexer Device Selector Examples

Multiplexer Channel	Device Selector	Example I/O Statement
interface select code 10, channel 0	1000	OUTPUT 1000;"Data" ENTER 1000;Number
interface select code 16, channel 3	1603	OUTPUT 1603;"Data" ENTER 1603;Number

I/O Path Names

An I/O path name is a data type that describes an I/O resource. With this BASIC system, you can assign I/O path names to:

Devices	ASSIGN @Device TO 722
HP-UX pipes	ASSIGN @Device TO " lp" (BASIC/UX only) (See the subsequent section called "Using HP-UX Pipes" for details.)
Files	ASSIGN @File TO "MyFile" (See the "Data Storage and Retrieval" chapter for details.)
Buffers	ASSIGN @Buff TO Buffer\$ (See the "Transfers and Buffered I/O" chapter of this manual for details.)

An I/O path name is any valid name preceded by the "@" character. A **name** is a combination of 1 to 15 characters, beginning with an uppercase alphabetical character or one of the characters CHR\$(161) through CHR\$(254) and followed by up to 14 lowercase alphanumeric characters, the underbar character (_), or the characters CHR\$(161) through CHR\$(254). Numeric-variable names are examples of valid names.

Assigning names to I/O paths provides many improvements in performance and additional capabilities over using device selectors, described in "Benefits of Using I/O Path Names" at the end of this chapter.

More Examples

ASSIGN @Display TO 1
OUTPUT @Display;"Data"

Assigns the I/O path name @Display to the CRT. Sends characters to the display.

ASSIGN @Printer TO 701
OUTPUT @Printer;"Data"

Assigns @Printer to HP-IB device 701. Sends characters to the printer.

ASSIGN @Printer TO "| lp"

Assigns @Printer to a pipe connected to the HP-UX lp command (the "line printer" spooler). (BASIC/UX only)

OUTPUT @Printer;"Data"

Sends characters to the spooler file. (See "Using HP-UX Printer Spoolers" in the chapter called "Using a Printer." for further information.)

14

ASSIGN @Serial TO 9
ENTER @Serial;String\$

Assigns @Serial to the interface at select code 9. Enters data from the serial interface into a string variable.

ASSIGN @Gpio TO 12
ENTER @Gpio;A_number

Assigns @Gpio to the interface at select code 12. Enters one numeric value from the interface.

ASSIGN @Window TO 601
PRINTER IS 601

Assigns @Window to window number 601. Assigns the window to be the "system printer". (BASIC/UX only)

ASSIGN @Date_pipe TO "date |"

Assigns the I/O path name @Date_pipe to a pipe connected to the HP-UX command named **date**. (BASIC/UX only)

ENTER @Date_pipe;Date_info\$

Enters output from the **date** command into the BASIC string named **Date_info\$**. (BASIC/UX only)

I/O Path Variable Contents. Since an I/O path name is a data type, a fixed amount of memory is allocated, or “reserved”, for the variable similar to the manner in which memory is allocated for other program variables (INTEGER, REAL, COMPLEX, and string variables). Since the variable does not initially contain usable information, the validity flag, shown below, is set to false. When the ASSIGN statement is actually executed, the allocated memory space is then filled with information describing the I/O path between the computer and the specified resource, and the validity flag is set to true. The I/O path contents are the following:

- Validity flag
- Type of resource
- Device selector of resource
- Additional information, if any, depends on the type of resource

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. However, 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 the memory space was allocated but the validity flag is still false; no valid information has been placed into the variable since the I/O path name has not yet been assigned to a resource.

This I/O path information is only accessible to the context in which it was allocated, unless it is passed as a parameter or appears in the proper COM statements. (See the *HP BASIC Language Reference* for details.) Thus, an I/O path name cannot be initially assigned from the keyboard, and it cannot be accessed from the keyboard unless it is presently assigned within the current context. However, an I/O path name can be re-assigned from the keyboard, as described in the next section.

This information describing the I/O path is accessed by the operating system whenever the I/O path name is specified in subsequent I/O statements. A portion of this information can also be accessed with the STATUS and CONTROL statements described in the “Registers” chapter. For now, the important point is that it contains a description of the resource sufficient to allow its access.

Re-Assigning I/O Path Names. If an I/O path name already assigned to a resource is to be re-assigned to another resource, the preceding form of the ASSIGN statement is also used. The resultant action is that the validity flag is first set false, implicitly “closing” the I/O path name to the device. A “new assignment” is then made just as if the first assignment never existed. Making this new assignment places information describing the specified device into the variable and sets the validity flag true. An example is shown below.

```
100  ASSIGN @Printer TO 1    ! Initial assignment.
110  OUTPUT @Printer;"Data1"
120  !
130  ASSIGN @Printer TO 701 ! 2nd ASSIGN closes 1st
140  OUTPUT @Printer;"Data2" ! and makes a new assignment.
150  PAUSE
160  END
```

14

The result of running the program is that “Data1” is sent to the CRT, and “Data2” is sent to 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 re-assigned 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. When the name is closed, the validity flag is set false, labeling the information as invalid. (Additional action may also be taken when the I/O path name assigned to a mass storage file is closed.) Attempting to use the closed name results in error 177 (**Undefined I/O path name**). Examples of statements that close path names are as follows.

Examples

```
ASSIGN @Printer TO *
ASSIGN @Serial TO *
ASSIGN @Window TO *    BASIC/UX only
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 re-assigned. This same name can be assigned either to the same or to a different resource with a subsequent ASSIGN statement. However, if it is used prior to being re-assigned, error 177 occurs.

I/O Path Names in Subprograms

When a subprogram or user-defined function is called, the context is changed to that of the called subprogram. The statements in the subprogram have access only to the data of the new context. Thus, in order to use an I/O path name in any statement within a subprogram, *one* of the following conditions must be true:

- The I/O path name must already be assigned within the context (i.e., the same instance of the subprogram).
- The I/O path name must be assigned in another context and passed to this context by reference as a parameter (i.e., specified in both the formal-parameter and pass-parameter lists).
- The I/O path name must be declared in a variable common (with COM statements) and already be assigned within a context that has access to that common block.

Subprograms and user-defined functions are discussed in detail in the “Subprograms and User-Defined Functions” chapter.

Benefits of Using I/O Path Names

Devices can be accessed with both device selectors and I/O path names, as shown in the previous discussions. With the information presented thus far, you may not see much difference between using these two methods of accessing devices. This section describes these differences in order to help you decide which method may be better for your application.

Execution Speed

When a device selector is used in an I/O statement to specify the I/O path to a device, the numeric expression must be evaluated by the computer every time the statement is executed. If the expression is complex, this evaluation might take several milliseconds.

```
OUTPUT Value_1+BIT(Value_2,5)*2^3;"Data"  
      | Device selector expression |
```

If a numeric variable is used to specify the device selector, this expression-evaluation time is reduced; this is the fastest execution possible when using

device selectors. However, more information about the I/O process must be determined before it can be executed.

In addition to evaluating the numeric expression, the computer must determine which type of interface (HP-IB, GPIO, etc.) is present at the specified select code. Once the type of interface has been determined, the corresponding attributes of the I/O path must then be determined before the computer can use the I/O path. Only after all of this information is known can the process of actually copying the data be executed.

If an I/O path name is specified in an OUTPUT or ENTER statement, all of this information has already been determined at the time the name was assigned to the I/O path. Thus, an I/O statement containing an I/O path name executes slightly faster than using the corresponding I/O statement containing a device selector (for the same set of source-list expressions).

Re-Directing Data

14

Using numeric-variable device selectors, as with I/O path names, allows a single statement to be used to move data between the computer and several devices. Simple examples of re-directing data in this manner are shown in the following programs.

Example of Re-Directing with Device Selectors

```
100 Device=1
110 GOSUB Data_out
    .
    .
    .
200 Device=9
210 GOSUB Data_out
    .
    .
    .
410 Data_out: OUTPUT Device;Data$
420 RETURN
```

Example of Re-Directing with I/O Path Names

```
100  ASSIGN @Device TO 1
110  GOSUB Data_out
    .
    .
    .
200  ASSIGN @Device TO 9
210  GOSUB Data_out
    .
    .
    .
410  Data_out: OUTPUT @Device;Data$
420  RETURN
```

The preceding two methods of re-directing data execute in approximately the same amount of time. As a comparison of the two methods, executing the “Device=” statement takes less time than executing the “ASSIGN @Device” statement. Conversely, executing the “OUTPUT Device” statement takes more time than executing the “OUTPUT @Device”. However, the overall time for each method is approximately equal.

There are two additional factors to be considered. First, device selectors cannot be used to direct data to mass storage files; I/O path names are the only access to files. If the data is ever to be directed to a file, you must use I/O path names. A good example of re-directing data to mass storage files is given in the “I/O Path Attributes” chapter. The second additional factor is described below.

Attribute Control

I/O paths have certain “attributes” which control how the system handles data sent through the I/O path. For example, the FORMAT attribute possessed by an I/O path determines which data representation will be used by the path during communications. If the path possesses the attribute of FORMAT ON, the ASCII data representation will be used. This is the default attribute automatically assigned by the computer when I/O path names are assigned to device selectors. If the I/O path possesses the attribute of FORMAT OFF, the internal data representation is used; this is the default format for BDAT files. Further details of these and additional attributes are discussed in the “I/O Path Attributes” chapter.

The second additional factor that favors using I/O path names is that you can control which attribute(s) are to be assigned to the I/O path to devices (and also to the I/O paths to files and buffers). If device selectors are used, this control is not possible. The “I/O Path Attributes” chapter describes how to specify the attributes to be assigned to an I/O path and gives several useful techniques for using the available attributes.

Outputting Data

Introduction

The preceding chapter described how to identify a specific device as the destination of data in an OUTPUT statement. Even though a few example statements were shown, the details of how the data are sent were not discussed. This chapter describes the topic of outputting data to devices. Outputting data to string variables, buffers, and mass storage files is described in the “I/O Path Attributes,” “Transfers and Buffered I/O,” and “Data Storage and Retrieval” chapters of this manual, and in the *HP BASIC Language Reference*.

There are two general types of output operations. The first type, known as **free-field outputs**, use the computer’s default data representations. For example, the ASCII representation described briefly in the preceding chapter is the default data representation used when communicating with devices, although the internal representation can also be used. (See the “I/O Path Attributes” chapter for further details.) 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.

15

Free-Field Outputs

Free-field outputs are invoked when the following types of OUTPUT statements are executed.

Examples

```
OUTPUT @Device;3.14*Radius^2
```

```
OUTPUT Printer;"String data";Num_1
```

```
OUTPUT 9;Test,Score,Student$
```

```
OUTPUT Escape_code$;CHR$(27)"&A1S";
```

The Free-Field Convention

The term **free-field** refers to the number of characters used to represent a data item. During free-field outputs, 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

15

The default data representation for devices is to use ASCII characters to represent numbers. The ASCII representation of each expression in the source list is generated during free-field output operations. Even though all REAL numbers have 15 (and INTEGERS can have up to 5) significant decimal digits of accuracy, not all of these digits are output with free-field OUTPUT statements. Instead, the following rules of the free-field convention are used when generating a number's ASCII representation.

All numbers between $1E-5$ and $1E+6$ are rounded to 12 significant digits and output in floating-point notation with no leading zeros. If the number is positive, a leading space is output for the sign; if negative, a leading “-” is output.

Examples

```
 32767  
-32768  
123456.789012  
-.000123456789012
```

15-2 Outputting Data

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.

Examples

```
-1.23456789012E+6  
1.23456789012E-5
```

Standard String Format

No leading or trailing spaces are output with the string’s characters.

String characters.
No leading or trailing spaces.

Note



The above 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.

15

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.

Sequence	Output data	Comment
1	<i>item 1</i>	
2	<i>item terminator</i>	Optional
3	<i>item 2</i>	
4	<i>item terminator</i>	Optional
<i>n</i>	<i>last item</i>	
<i>n+1</i>	<i>EOL Sequence</i>	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.

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

15

i	t	e	m	CR	LF	-	1	2	3	4	EOL sequence
---	---	---	---	----	----	---	---	---	---	---	-----------------

The default EOL sequence is a CR/LF

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

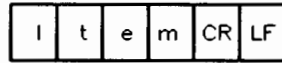
```
OUTPUT Device;-1234,"Item"
```

-	1	2	3	4	,	i	t	e	m	EOL sequence
---	---	---	---	---	---	---	---	---	---	-----------------

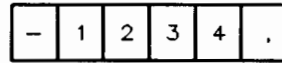
15-4 Outputting Data

If a separator follows the last item in the list, the proper item terminator will be output *instead* of the EOL sequence.

```
OUTPUT Device;"Item",
```

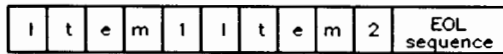


```
OUTPUT Device;-1234,
```

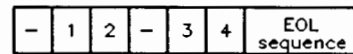


Using a **semicolon separator** suppresses output of the (otherwise automatic) item's terminator.

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

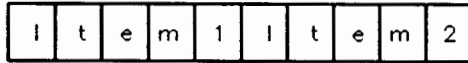


```
OUTPUT 1;-12;-34
```



If a semicolon separator follows the last item in the list, the EOL sequence and item terminators are suppressed.

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



Neither of the item 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

```

Resultant Output

15

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.

15-6 Outputting Data

```

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

```



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																		

15

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 the “I/O Path Attributes” chapter 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 device-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 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. The individual chapter that describes programming each interface further describes each interface's END indication (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.

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.

15-8 Outputting Data

Using END in Freefield OUTPUT

The secondary keyword END may be optionally specified following the last source-item expression in a freefield OUTPUT statement. The result is to *suppress the End-of-Line (EOL) sequence* that would otherwise be output after the last byte of the last source item. If a comma is used to separate the last item from the END keyword, the corresponding item terminator will be output as before (carriage-return and line-feed for string items and comma for numeric items).

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
---	---	---

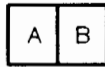
Neither item terminator nor EOL sequence is sent.

```
OUTPUT @Gpio;"AB",END
```

A	B	CR	LF
---	---	----	----

Item terminator, but no EOL sequence, is sent.

```
OUTPUT @Gpio;"AB";END
OUTPUT @Gpio;"AB" END
```



Neither item terminator nor EOL sequence is sent.

```
OUTPUT @Gpio
```



The EOL sequence is sent.

```
OUTPUT @Gpio;END      No EOL sequence sent
OUTPUT @Gpio;" " END  No EOL sequence sent
```

The END keyword has additional significance when the destination is a mass storage file. See the “Data Storage and Retrieval” chapter for further details.

15

Additional Definition

BASIC defines additional action when END is specified in a freefield OUTPUT statement directed to either HP-IB or Data Communications interfaces.

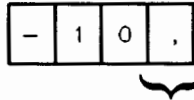
END with HP-IB Interfaces

With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the last 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” chapter in the *HP BASIC 6.2 Interface Reference* manual.

Examples

```
ASSIGN @Device TO 701
```

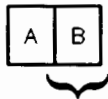
```
OUTPUT @Device;-10,END
```



EOI sent with the last character
(numeric item terminator).

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

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



EOI sent with the last character of the item.

```
OUTPUT @Device;END      Neither EOL sequence nor EOI is sent, since  
OUTPUT @Device;"" END  no data is sent.
```

END with the Data Communications Interface

With Data Communication interfaces, END has the additional function of sending an end-of-data indication to the interface. See the "Datacomm Interface" chapter in the *HP BASIC 6.2 Interface Reference* manual for further details.

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
```

15

Images

Images are used to specify the format of data during I/O operations. Each image consists of groups of individual image (or “field”) specifiers, such as 6A, SDDD.DDD, and 3X in the preceding examples. Each of these field specifiers describe one of the following things:

- It describes the desired format of one item in the source list. (For instance, 6A specifies that a string item is to be output in a “6-character Alpha” field. SDDD.DDD specifies that a numeric item is to be output with Sign, 3 Decimal digits preceding the decimal point, followed by 3 Decimal digits following the decimal point.)
- It specifies that special character(s) are to be output. (For instance, 3X specifies that 3 spaces are to be output.) There is no corresponding item in the source list.

Thus, you can think of the image list as either a precise format description or as a procedure. It is convenient to talk about the image list as a procedure for the purpose of explaining how this type of OUTPUT statement is executed.

Again, each image list consists of images that each describe the format of data item to be output. The order of images in the list corresponds to the order of data items in the source list. In addition, image specifiers can be added to output (or to suppress the output of) certain characters. The following example steps through exactly how BASIC executes all of the preceding equivalent statements.

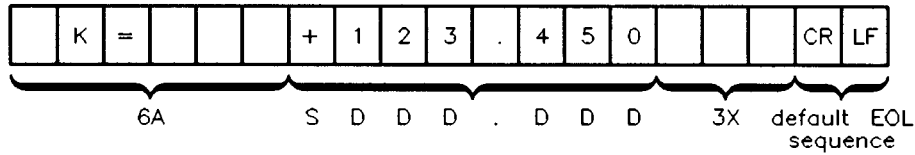
15

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.



15

- Step 1. The computer evaluates the first image in the list. Generally, each group of specifiers separated by commas is an “image”; the commas tell the computer that the image is complete and that it can be “processed”. In general, each group of specifiers is processed before going on to the next group. In this case, six alphanumeric characters taken from the first item in the source list are to be output.
- Step 2. The computer then evaluates the first item in the source list and begins outputting it, one byte (or word) at a time. After the fourth character, the first expression has been “exhausted”. In order to satisfy the corresponding specifier, two spaces (alphanumeric “fill” characters) are output.
- Step 3. The computer evaluates the next image (note that this image consists of several different image specifiers). The “S” specifier requires that a sign character be output for the number, the “D” specifiers require digits of a number, and the “.” specifies where the decimal point will be placed. Thus, the number of digits following the decimal point have been specified. All of these specifiers describe the format of the next item in the source list.
- Step 4. The next data item in the source list is evaluated. The resultant number is output one digit at a time, according to its image specifiers. A trailing zero has been added to the number to satisfy the “DDD” specifiers following the decimal point.
- Step 5. The next image in the list (“3X”) is evaluated. This specifier does not “require” data, so the source list needs no corresponding expression. Three spaces are output by this image.

15-14 Outputting Data

Step 6. Since the entire image list and source list have been “exhausted”, the computer then outputs the current (or default, if none has been specified) “end-of-line” sequence of characters (here we assume that a carriage-return and line-feed are the current EOL sequence).

The execution of the statement is now complete. As you can see, the data specified in the source list must match those specified in the output image in type and in number of items.

Image Definitions During Outputs

This section describes the definitions of each of the image specifiers when referenced by OUTPUT statements. The specifiers have been categorized by data type. It is suggested that you scan through the description of each specifier and then look over the examples. You are also highly encouraged to experiment with the use of these concepts.

Numeric Images

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

15

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.

15

Sign, Digit, Radix and Exponent Specifiers (continued)

Image Specifier	Meaning
E	Specifies that the number is to be output using scientific notation. The "E" must be preceded by at least one digit specifier (D, Z, or *). The default exponent is a four-character sequence consisting of an "E", the exponent sign, and two exponent digits, equivalent to an "ESZZ" image. Since the number is left-justified in the specified digit field, the image for a negative number must contain a sign specifier (see the next section).
ESZ	Same as "E" but only 1 exponent digit is output.
ESZZZ	Same as "E" but three exponent digits are output.
K, -K	Specifies that the number is to be output in a "compact" format, similar to the standard numeric format; however, neither leading spaces (that would otherwise replace a "+" sign) nor item terminators (commas) are output, as would be with the standard numeric format.
H, -H	Like K, except that the number is to be output using a comma radix (European radix).

Numeric Examples

15

OUTPUT @Device USING "DDDD";-123.769

-	1	2	4	EOL sequence
---	---	---	---	-----------------

OUTPUT @Device USING "4D";-1.2

-	1	EOL sequence
---	---	-----------------

OUTPUT @Device USING "ZZ.DD";1.675

0	1	.	6	8	EOL sequence
---	---	---	---	---	-----------------

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

0	.	4	EOL sequence
---	---	---	-----------------

OUTPUT @Device USING "DD.E";12345

1	2	.	E	+	0	3	EOL sequence
---	---	---	---	---	---	---	-----------------

15

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

2	0	.	0	0	E	-	0	5	EOL sequence
---	---	---	---	---	---	---	---	---	-----------------

OUTPUT @Device USING "K";12.400

1	2	.	4	EOL sequence
---	---	---	---	-----------------

15-18 Outputting Data

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

-	1	2	.	4	5	EOL sequence
---	---	---	---	---	---	-----------------

OUTPUT CRT USING "MDD.DD";2.09

		2	.	0	9	EOL sequence
--	--	---	---	---	---	-----------------

OUTPUT 1 USING "SD.D";2.449

+	2	.	4	EOL sequence
---	---	---	---	-----------------

OUTPUT 1 USING "SZ.DD";.49

+	0	.	4	9	EOL sequence
---	---	---	---	---	-----------------

15

OUTPUT CRT USING "SDD.DDE";-2.35

-	2	3	.	5	0	E	-	0	1	EOL sequence
---	---	---	---	---	---	---	---	---	---	-----------------

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

*	2	.	6	EOL sequence
---	---	---	---	-----------------

OUTPUT @Device USING "DRDD";3.1416

3	.	1	4	EOL sequence
---	---	---	---	-----------------

OUTPUT @Device USING "H";3.1416

3	.	1	4	1	6	EOL sequence
---	---	---	---	---	---	-----------------

15

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 byte is to be output as a character ¹ . Trailing spaces are used as fill characters if the string contains less than the number of characters specified.
"literal"	All characters placed in quotes form a string literal, which is output exactly as is. Literals can be placed in output images which are part of OUTPUT statements by enclosing them in double quotes.
K, -K, H, -H	Specifies that the string is to be output in "compact" format, similar to the standard string format; however, no item terminators are output as with the standard string format.

¹If you are using the Extended ASCII character set, each character is one byte long. If you are using a localized version of BASIC that supports a two-byte language, such as Japanese, characters can be two bytes long. Use AA to print a two-byte character. Refer to the *HP BASIC 6.2 Porting and Globalization* manual for details about two-byte characters.

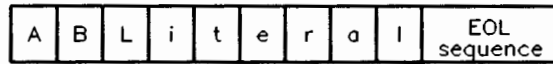
15

String Examples

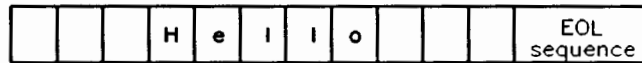
```
OUTPUT @Device USING "8A";"Characters"
```

C	h	a	r	a	c	t	e	EOL sequence
---	---	---	---	---	---	---	---	-----------------

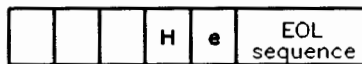
OUTPUT @Device USING "K","Literal";"AB"



OUTPUT @Device USING "K";" Hello "



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



15

Binary Images

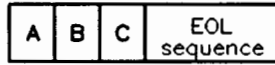
These image specifiers are used to output bytes (8-bit data) and words (16-bit data) to the destination. Typical uses are to output non-ASCII characters or integers in their internal representation.

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	<p>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.</p> <p>If either an I/O path name with the BYTE attribute (see the "I/O Path Attributes" chapter) 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.</p> <p>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.</p> <p>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 (see the "I/O Path Attributes" chapter for details).</p>
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).

Binary Examples

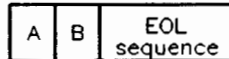
OUTPUT @Device USING "B,B,B";65,66,67



OUTPUT @Device USING "B";13



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



15

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

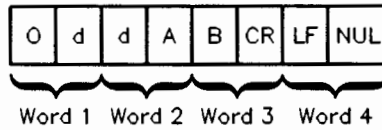


Word 1 Word 2 Word 3 Word 4 Word 5

END Indication Sent Here

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



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.

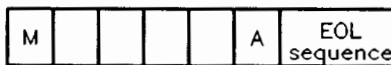
Special-Character Specifiers

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.

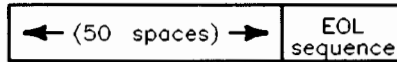
15

Special-Character Examples

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



OUTPUT @Device USING "50X"



OUTPUT @Device USING "@,/"



OUTPUT @Device USING "/"



15

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).

15

Termination Examples

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



D	a	t	a	EOL sequence	EOL sequence
---	---	---	---	-----------------	-----------------

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

D	a	t	a
---	---	---	---

OUTPUT @Device USING "#,B";12

FF

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

D	a	t	a	CR
---	---	---	---	----

15

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

EOL sequence	D	a	t	a	LF
-----------------	---	---	---	---	----

Additional Image Features

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

Repeat Factors

Many of the specifiers can be repeated without having to explicitly list the specifier as many times as it is to be repeated. For instance, to a character field of 15 characters, you do not need to use "AAAAAAAAAAAAAAAAA"; instead, you merely specify the number of times that the specifier is to be repeated in front of the image ("15A"). The following specifiers can be repeated by specifying an integer repeat factor; the specifiers not listed cannot be repeated in this manner.

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

Examples

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

0	3	2	8	.	0	3	0	EOL sequence
---	---	---	---	---	---	---	---	-----------------

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

D	a	t	a		b	EOL sequence
---	---	---	---	--	---	-----------------

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

					D	a	EOL sequence
--	--	--	--	--	---	---	-----------------

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

EOL sequence	EOL sequence	D	a	t	a	EOL sequence
-----------------	-----------------	---	---	---	---	-----------------

OUTPUT @Device USING "8A,20";"The End"

T	h	e		E	n	d		FF	FF	EOL sequence
---	---	---	--	---	---	---	--	----	----	-----------------

OUTPUT @Device USING "2/"

CR	LF	CR	LF	EOL sequence
----	----	----	----	-----------------

15 Image Re-Use

If the number of items in the source list exceeds the number of matching specifiers in the image list, the computer attempts to re-use the image(s) beginning with the first image.

```
110 ASSIGN @Device TO CRT
120 Num_1=1
130 Num_2=2
140 !
150 OUTPUT @Device USING "K";Num_1,"Data_1",Num_2,"Data_2"
160 OUTPUT @Device USING "K,/";Num_1,"Data_1",Num_2,"Data_2"
170 END
```

Resultant Display

```
1Data_12Data_2
1
Data_1
2
Data_2
```

Since the “K” specifier can be used with both numeric and string data, the above OUTPUT statements can re-use the image list for all items in the source list. If any item cannot be output using the corresponding image item, an error results. In the following example, “Error 100 in 150” occurs due to data mismatch.

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

Nested Images

Another convenient capability of images is that they can be nested within parentheses. The entire image list within the parentheses will be used the number of times specified by the repeat factor preceding the first parenthesis. The following program is an example of this feature.

```
100 ASSIGN @Device TO 701
110 !
120 OUTPUT @Device USING "3(B),X,DD,X,DD";65,66,67,68,69
130 END
```

15

Resultant Output

A	B	C		6	8		6	9	EOL sequence
---	---	---	--	---	---	--	---	---	-----------------

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

END with OUTPUTs that Use Images

Using the optional secondary keyword END in an OUTPUT statement that uses an image produces results which differ from those of using END in a freefield OUTPUT statement. Instead of always suppressing the EOL sequence, the END keyword *only suppresses the EOL sequence when no data are output from the last source-list expression*. Thus, the “#” image specifier generally controls the suppression of the otherwise automatic EOL sequence, while the END keyword suppresses it only in less common usages.

15

Examples

Device=12

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

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

OUTPUT Device USING "K";"ABC" END

A	B	C	EOL sequence
---	---	---	-----------------

The EOL sequence is not suppressed.

OUTPUT Device USING "L,/,""Literal"" ,X,0"

EOL sequence	CR	LF	L	i	t	e	r	a	l		FF	EOL sequence
-----------------	----	----	---	---	---	---	---	---	---	--	----	-----------------

In this case, specifiers that require no source-item expressions are used to generate characters for the output; there are no source expressions. The EOL sequence is output after all specifiers have been used to output their respective characters. Compare this action to that shown in the next example.

OUTPUT Device USING "L,/,""Literal"" ,X,0;END

EOL sequence	CR	LF	L	i	t	e	r	a	l		FF
-----------------	----	----	---	---	---	---	---	---	---	--	----

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

15

Additional END Definition

The END secondary keyword has been defined to produce additional action when included in an OUTPUT statement directed to HP-IB and Data Communications interfaces.

END with HP-IB Interfaces

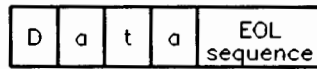
With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the *last character* of either the last source item or the EOL sequence (if sent). As with freefield OUTPUT, *no EOI is sent if no data is sent from the last source item and the EOL sequence is suppressed.*

Examples

```
ASSIGN @Device TO 701
```

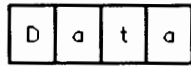
```
OUTPUT @Device USING "K";"Data",END
```

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



EOI sent with last character
of the EOL sequence.

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



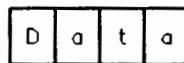
EOI sent with this character.

15

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
```



The EOI was not sent in either case, since no data were sent from the last source item *and* the EOL sequence was suppressed.

END with Data Communications Interfaces

With Data Communications interfaces, END has the additional definition of sending an end-of-data indication to the interface in the same instances in which EOI would be sent on HP-IB interfaces. See the "Datacomm Interface" chapter in the *HP BASIC 6.2 Interface Reference* manual for further details.

Entering Data

This chapter discusses the topic of entering data from devices. You may already be familiar with the OUTPUT statement described in the previous chapter; many of those concepts are applicable to the process of entering data. Earlier in this manual, you were told that *the data output from the sender had to match that expected by the receiver*. Because of the many ways that data is represented in external devices, entering data can sometimes require more programming skill than outputting data. In this chapter, you will see what is involved in being the receiving device. Both free-field enters and enters that use images are described, and several examples are given with each topic.

Free-Field Enters

Executing the free-field form of the ENTER invokes conventions which are the “converse” of those used with the free-field OUTPUT statement. In other words, data output using the free-field form of the OUTPUT statement can be readily entered using the free-field ENTER statement; no explicit image specifiers are required. The following statements exemplify this form of the ENTER statement.

Examples

```
ENTER @Voltmeter;Reading

ENTER 724;Readings(*)

ENTER From_string$;Average,Student_name$

ENTER @From_file;Data_code,Str_element$(X,Y)
```

Item Separators

Destination items in ENTER statements can be separated by *either* a comma or a semicolon. Unlike the OUTPUT statement, it makes *no difference* which is used; data will be entered into each destination item in a manner independent of the punctuation separating the variables in the list. However, *no trailing punctuation is allowed*. The first two of the following statements are equivalent, but an error is reported when the third statement is executed.

Examples

```
ENTER @From_a_device;N1,N2,N3  These first two statements are equivalent.  
ENTER @From_a_device;N1;N2;N3
```

```
ENTER @From_a_device;N1,N2,N3,  Executing this statement causes an error (because  
of trailing comma).
```

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.

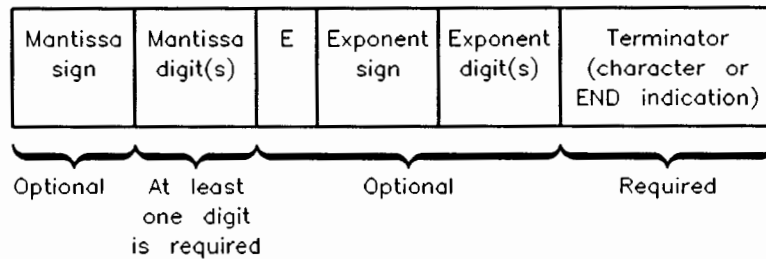
In addition to the termination characters, each item can be terminated (only with selected interfaces) by a device-dependent END indication. For instance, some interfaces use a signal known as EOI (End-or-Identify). The EOI signal is only available with the HP-IB, CRT, and keyboard interfaces. EOI termination 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). See the "I/O Path Attributes" chapter 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.



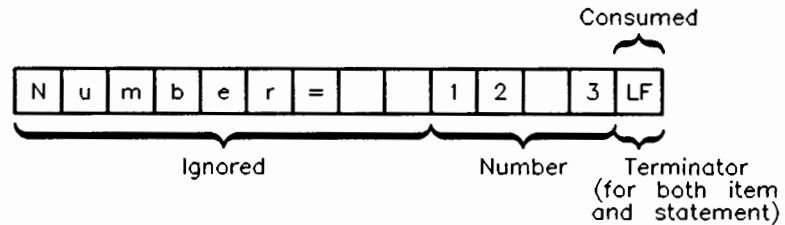
Numeric characters include decimal digits “0” through “9” and the characters “.”, “+”, “-”, “E”, and “e”. These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of the number, it will be treated as a non-numeric character.

The following rules are used by the number builder to construct numbers from incoming streams of ASCII numeric characters.

1. All leading non-numeric characters are ignored; all leading and imbedded spaces are ignored.

Example

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

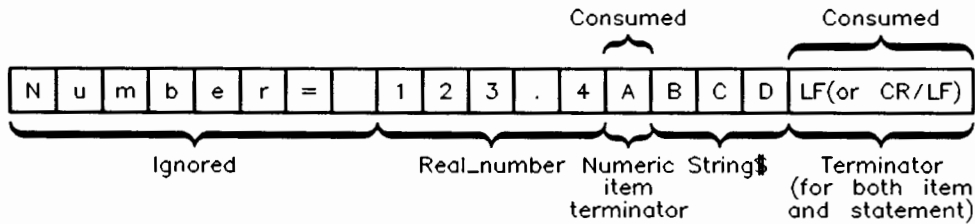


The result of entering the preceding data with the given ENTER statement is that Number receives a value of 123. The line-feed (statement terminator) is *required* since Number is the last item in the destination list.

2. Trailing non-numeric characters terminate entry into a numeric variable, and the terminating characters (of *both* string and numeric items) are “consumed”. In this manual, “consumed” characters refers to characters *used to terminate* an item but not entered into the variable; “ignored” characters are entered but are *not used*.

Example

`ENTER @Device;Real_number,String$`

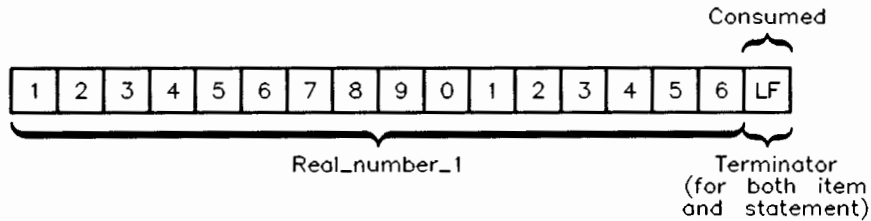


The result of entering the preceding data with the given ENTER statement is that Real_number receives the value 123.4 and String\$ receives the characters “BCD”. The “A” was lost when it terminated the numeric item; the string-item terminator(s) are also lost. The string-item terminator(s) also terminate the ENTER statement, since String\$ is the last item in the destination list.

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

Example

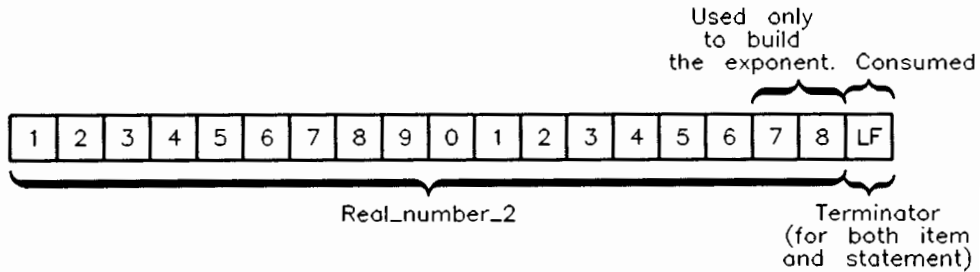
ENTER @Device;Real_number_1



The result of entering the preceding data with the given ENTER statement is that Real_number_1 receives the value 1.234567890123456 E+15.

Example

ENTER @Device;Real_number_2



16

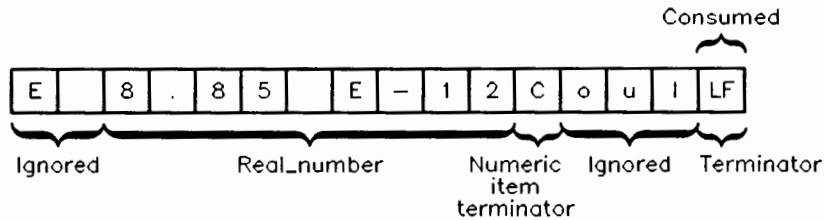
The result of entering the preceding data with the given ENTER statement is that Real_number_2 receives the value 1.234567890123456 E+17.

16-6 Entering Data

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.

Example

`ENTER @Device;Real_number`



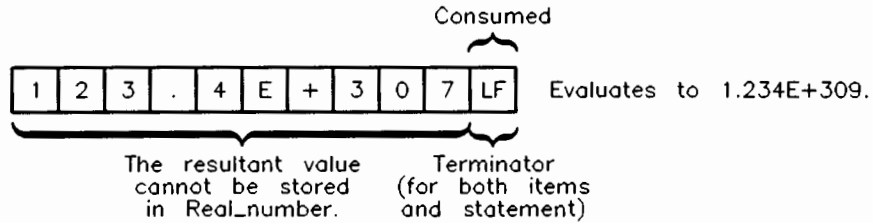
The result of entering the preceding data with the given ENTER statement is that Real_number receives a value of 8.85 E-12. The character C terminates entry into Real_number, and the characters ou l are entered (but ignored) in search of the required line-feed statement terminator. If the character C is to be entered but not ignored, you must use an image. Using images with the ENTER statement is described later in this chapter.



- 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.

Example

ENTER @Device;Real_number



The data is entered but evaluates to a number outside the range of REAL numbers. Consequently, error 19 is reported, and the variable *Real_number* retains its former value.

- If the item is the *last* one in the list, *both* the item and the statement need to be properly *terminated*. If the numeric item is terminated by a non-numeric character, the statement will *not* be terminated until it either receives a line-feed character or an END indication (such as EOI signal with a character). The topic of terminating free-field ENTER statements is described later in this chapter in the section of the same name.

16

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

16-8 Entering Data

(during free-field enters); they are “lost” when they terminate the entry. The EOI signal also terminates entry into a string variable, but the variable must be the last item in the destination list (during free-field enters).

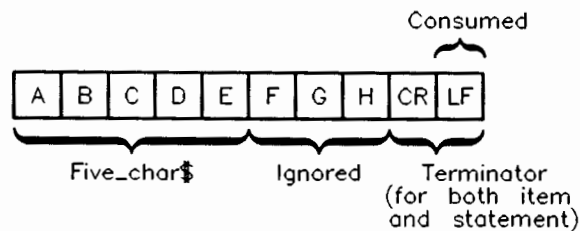
All characters received from the source are entered directly iemph appropriate string variable until *any* of the following conditions occurs:

- an item terminator character is received.
- the number of characters entered equals the dimensioned length of the string variable.
- the EOI signal is received.

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

Example

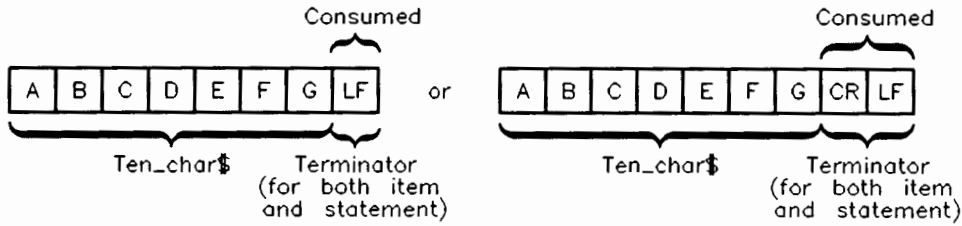
```
ENTER @Device;Five_char$
```



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).

Example

ENTER @Device;Ten_char\$



The result of entering the preceding data with the given ENTER statement is that Ten_char\$ receives the characters ABCDEFG and the terminating LF (or CR/LF) is lost.

The following example illustrates possible interactions between the two terminating conditions for strings.

Example

```
10 DIM A$(5),B$(4),C$(4),D$(4),E$(4),F$(4),G$(4),H$(4)
20 ASSIGN @File TO "File";FORMAT ON
30 ENTER @File;A$,B$,C$,D$,E$,F$,G$,H$
40 DISPLAY FUNCTIONS ON
50 PRINT A$&"|"&B$&"|"&C$&"|"&D$&"|"&E$&"|"&F$&"|"&G$&"|"&H$;
60 DISPLAY FUNCTIONS OFF
70 END
```

16

The file called File contains the data:

A	B	C	CR	LF	E	F	G	H	CR	LF	I	J	K	L	M	N	LF	O	P	Q	R	S	T	U	V	W	X	Y	Z
---	---	---	----	----	---	---	---	---	----	----	---	---	---	---	---	---	----	---	---	---	---	---	---	---	---	---	---	---	---

When RUN, this program prints:

```
ABCDCR  
|EFGH|JKL|MN|OPQR|STUV
```

which shows that: **A\$** received the first five characters and entry to the item was terminated because the string variable was filled completely; **B\$** terminated on the line feed; **C\$** received **EFGH** and terminated; **D\$** terminated on carriage return/line feed; **E\$** received the next four characters; **F\$** terminated on carriage return/line feed after receiving two characters; and **G\$** and **H\$** took four characters each.

The important points here are:

- carriage returns are not consumed unless the line feed would also fit into the string.
- no “scan ahead to the terminator” is performed except for the last variable in an ENTER statement.

In order to avoid the above behavior, 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, receiving EOI with a character, and receiving a control block while entering data through the Data Communications interface
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.

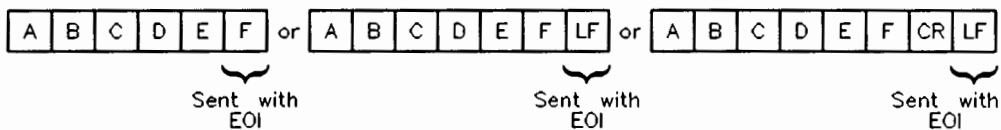
Another case involving numeric data can also occur (see the example given with “rule 4” describing the number builder). If a trailing non-numeric character terminates the last item (which is a numeric variable), additional characters will be entered in search of either a line-feed or a character accompanied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

Example

ENTER @Device;String\$



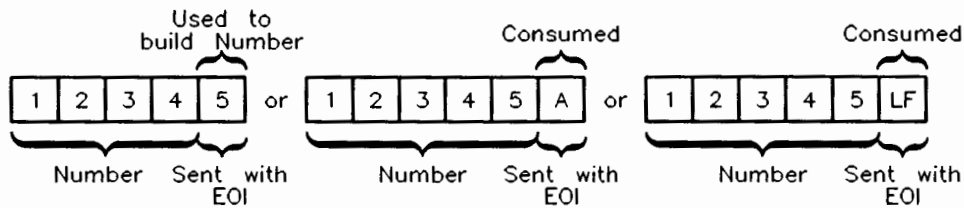
The result of entering the preceding data with the given ENTER statement is that String\$ receives the characters “ABCDEF”. The EOI signal being received

16-12 Entering Data

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.

Example

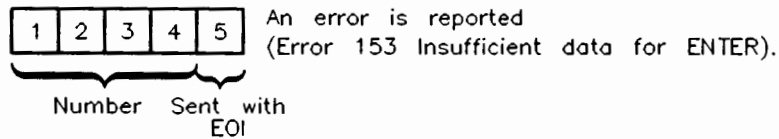
ENTER @Device;Number



The result of entering any of the above data streams with the given ENTER statement is that Number receives the value 12345. If the EOI signal accompanies a numeric character, it is entered and used to build the number; if the EOI is received with a numeric terminator, the terminator is lost as usual.

Example

ENTER @Device;Number,String\$



The result of entering the preceding data with the given statement is that an *error is reported* when the character 5 accompanied by EOI is received. However, Number receives the value 12345, but String\$ retains its previous value. An error is reported because *all* variables in the destination list have *not* been satisfied when the EOI is received. Thus, the EOI signal is an *immediate statement terminator during free-field enters*. The EOI signal has a *different* definition during enters that use images, as described later in this chapter.

The EOI signal is implemented on the HP-IB Interface, described in the “HP-IB Interface” chapter of the *HP BASIC 6.2 Interface Reference* manual. Since it is often convenient to use the keyboard and CRT for external devices, these internal devices have been designed to simulate this signal. Further descriptions of this feature’s implementation in the CRT display and keyboard are contained in the “Display Interfaces” and “Keyboard Interfaces” chapters of the *HP BASIC 6.2 Interface Reference* manual, respectively.

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.

16

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.

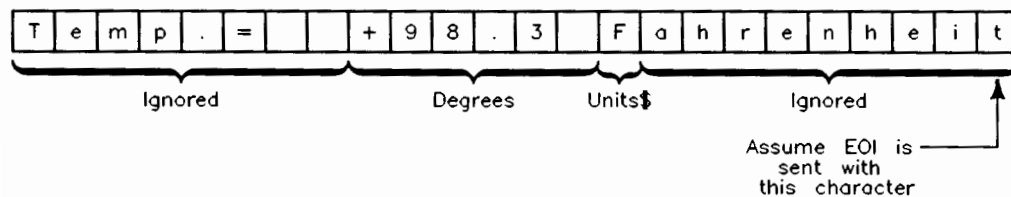
The examples given here treat the image list as a *procedure*.

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.



Given the preceding conditions, let's look at how the computer executes the following ENTER statement that uses the specified IMAGE statement.

```
300 ENTER @Device USING Image_1;Degrees,Units$  
310 Image_1: IMAGE 8X,SDDD.D,A
```

- Step 1. The computer evaluates the first image of the IMAGE statement. It is a special image in that it does not correspond to a variable in the destination list. It specifies that eight characters of the incoming data stream are to be ignored. Eight characters, **Temp.=** , are entered and are ignored (i.e., are not entered into any variable).
- Step 2. The computer evaluates the next image. It specifies that the next six characters are to be used to build a number. Even though the order of the sign, digit, and radix are explicitly stated in the image, the actual order of these characters in the incoming data stream does not have to match this specifier exactly. Only the *number* of numeric specifiers in the image, here six, is all that is used to specify the data format. When all six characters have been entered, the number builder attempts to form a number.
- Step 3. After the number is built, it is placed into the variable "Degrees"; the representation of the resultant number depends on the numeric variable's type (INTEGER, REAL, or COMPLEX). Note that 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.

16

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.

16-16 Entering Data

Image Definitions During Enter

This section describes the individual image specifiers in detail. The specifiers have been categorized into data and function type.

Numeric Images

Sign, digit, radix, and exponent specifiers are all used identically in ENTER images. The number builder can also be used to enter numeric data.

Numeric Specifiers

Image Specifier	Meaning
D	Specifies that one byte is to be entered and interpreted as a numeric character. If the characters is non-numeric (including leading spaces and item terminators), it will still "consume" one digit of the image item.
Z, *	Same action as D. Keep in mind that A and * can only appear to the left of the radix indicator (decimal point or R) in a numeric image item.
S, M	Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must follow either of these specifiers in an image item.
.	Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must accompany this specifier in an image item.
R	Same action as D in that one byte is to be entered and interpreted as a numeric character; however, when R is used in a numeric image, it directs the number builder to use the comma as a radix indicator and the period as a terminator to the numeric item. At least one digit specifier must accompany this specifier in the image item.

Numeric Specifiers (Continued)

Image Specifier	Meaning
E	Equivalent to 4D, if preceded by at least one digit specifier (Z, *, or D) in the image item. The following specifiers must also be preceded by at least one digit specifier.
ESZ	Equivalent to 3D.
ESZZ	Equivalent to 4D.
ESZZZ	Equivalent to 5D.
K, -K	Specifies that a variable number of characters are to be entered and interpreted according to the rules of the number builder (same rules as used in "free-field" ENTER operations).
H, -H	Like K, except that a comma is used as the radix indicator, and a period is used as the terminator for the numeric item.

Examples of Numeric Images

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.

16

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.

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 ¹ 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 <i>both</i> 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 the “I/O Path Attributes” chapter for further explanation.

¹If you are using the Extended ASCII character set, each character is one byte long. If you are using a localized version of BASIC that supports a two-byte language, such as Japanese, characters can be two-bytes long. Use AA to print a two-byte character. Refer to the *HP BASIC 6.2 Porting and Globalization* manual for details about two-byte characters.

Examples of String Images

ENTER @Device USING "10A";Ten_chars\$	Enter 10 characters.
ENTER @Device USING "K";Any_string\$	Enter using the 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.

16

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 6th through 9th characters.

16-20 Entering Data

ENTER @Device USING "/",K";String2\$

*Ignore 1st 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 the "I/O Path Attributes" chapter) 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 <i>not</i> overwritten (as with the W specifier).

16

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

16

This section describes the default statement-termination conditions for enters that use images (for devices). The effects of numeric-item and string-item terminators and the end-or-identify (EOI) signal during these operations are discussed in this section. After reading this section, you will be able to better understand how enters that use images work and how the default statement-termination conditions are *modified* by the #, %, +, and - image specifiers.

Default Termination Conditions

The default statement-termination conditions for enters that use images are very similar to those required to terminate free-field enters. *Either* of the following conditions will properly terminate an ENTER statement that uses an image.

- An END indication (such as the EOI signal or end-of-data) is received *with* the byte that satisfies the last *image item within 256 bytes after* the byte that satisfied the last image item.
- A line-feed is received *as* the byte that satisfies the last *image item* (exceptions are the “B” and “W” specifiers) or *within 256 bytes after* the byte that satisfied the last image item.

EOI Re-Definition

It is important to realize that when an enter uses an image (when the secondary keyword “USING” is specified), the definition of the EOI signal is *automatically modified*. If the EOI signal terminates the *last image item*, the entire statement is properly terminated, as with free-field enters. In addition, *multiple EOI signals are now allowed* and act as *item terminators*; however, the EOI must be received *with* the byte that satisfies each image item. If the EOI is received *before* any image is satisfied, it is *ignored*. Thus, all images must be satisfied, and EOI will not cause early termination of the ENTER-USING-image statement.

The following table summarizes the definitions of EOI during several types of ENTER statement. The statement-terminator modifiers are more fully described in the next section.

Effects of EOI During ENTER Statements

	Free-Field ENTER Statements	ENTER USING without # or %	ENTER USING with #	ENTER USING with %
Definition of EOI	Immediate statement terminator	Item terminator or statement terminator	Item terminator or statement terminator	Immediate statement terminator
Statement Terminator Required?	Yes	Yes	No	No
Early Termination Allowed?	No	No	No	Yes

Statement-Termination Modifiers

These specifiers modify the conditions that terminate enters that use images. The first one of these specifiers encountered in the image list modifies the termination conditions for the ENTER statement. If another of these specifiers is encountered in the image list, it again modifies the terminating conditions for the statement.

Statement-Termination Modifiers

Image Specifier	Meaning								
#	Specifies that a statement-termination condition is <i>not</i> required; the ENTER statement is automatically terminated as soon as the <i>last image item</i> is satisfied.								
%	<p>Also specifies that a statement-termination condition is not required. In addition, EOI is re-defined to be an <i>immediate</i> statement terminator, <i>allowing early termination</i> of the ENTER <i>before all</i> image items have been satisfied. However, the statement can only be terminated on a "legal item boundary". The legal boundaries for different specifiers are as follows:</p> <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; border-bottom: 1px solid black;">Specifier</th> <th style="text-align: left; border-bottom: 1px solid black;">Legal Boundary</th> </tr> </thead> <tbody> <tr> <td style="border-bottom: 1px solid black;">K, -K</td> <td style="border-bottom: 1px solid black;">With any character, since this specifies a variable-width field of characters.</td> </tr> <tr> <td style="border-bottom: 1px solid black;">S, M, D, E, Z, ., A, X, <i>literal</i>, B, W</td> <td style="border-bottom: 1px solid black;">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 style="border-bottom: 1px solid black;">/</td> <td style="border-bottom: 1px solid black;">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> </tbody> </table>	Specifier	Legal Boundary	K, -K	With any character, since this specifies a variable-width field of characters.	S, M, D, E, Z, ., A, X, <i>literal</i> , B, W	Only with the last character that satisfies the image (e.g., with the 5th character of a 5A image). If EOI is received with any other character, it is ignored.	/	Only with the last line-feed character that satisfies the image (e.g., with the 3rd line-feed of a "3/" image); otherwise it is ignored.
Specifier	Legal Boundary								
K, -K	With any character, since this specifies a variable-width field of characters.								
S, M, D, E, Z, ., A, X, <i>literal</i> , B, W	Only with the last character that satisfies the image (e.g., with the 5th character of a 5A image). If EOI is received with any other character, it is ignored.								
/	Only with the last line-feed character that satisfies the image (e.g., with the 3rd line-feed of a "3/" image); otherwise it is ignored.								
+	Specifies that an END indication is required to terminate the ENTER statement. Line-feeds are ignored as statement terminators; however, they will still terminate items (unless a -K or -H image is used for strings).								
-	Specifies that a line-feed is required to terminate the statement. EOI is ignored, and other END indications (such as EOF or end-of-data) cause an error if encountered before the line-feed.								

Examples of Modifying Termination Conditions

ENTER @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 it 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

16

All of the following specifiers can be preceded by an integer that specifies how many times the specifier is to be used.

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

Image Re-Use

If there are fewer images than items in the destination list, the list will be re-used, beginning with the first item in the image list. If there are more images than there are items, the additional specifiers will be ignored.

Examples

```
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.

Example

```
ENTER @Source USING "2(B,5A,/),/";N1,N1$,N2,N2$
```


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 chapter; however, the specific definitions of the interface registers are given in the *HP BASIC 6.2 Interface Reference* manual.

There are *three levels of register access*.

- 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.

```

- STATUS and CONTROL (firmware) registers are explicitly accessed by BASIC statements:

```

100 STATUS CRT,13;Crt_height
110 CONTROL CRT,13;Crt_height+3

```

- Interface (hardware) registers are directly read or written.

```

100 READIO 15,0;Card_id
110 WRITEIO 15,3;Intr_mask ! Write to Breadboard card reg. 3

```

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 ";".
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 *HP BASIC 6.2 Interface Reference* manual. The definitions of I/O path registers are described later in this chapter.

Example

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.

17

```
          Interface select code
          /
STATUS 2 , 7 ; Reg_7
          ^
Register number      Numeric variable(s)
(optional)           receive register(s) contents
```

17-2 Registers

Example

I/O path STATUS register 0 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
```

Example

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 *HP BASIC 6.2 Interface Reference* manual.

Example

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
          to be sent to register
          (register 0 in this case)
```


Example

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
(optional)

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.

17

- 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

17-4 Registers

path register 0 to determine the type of resource to which the I/O path name has been assigned.

```
710 Find_type: STATUS @Resource;Reg_0
720     IF Reg_0=0 THEN GOTO Not_assigned
730     IF Reg_0=1 THEN GOTO Device
740     IF Reg_0=2 THEN GOTO File
750     IF Reg_0=3 THEN GOTO Buffer
760     IF Reg_0=4 THEN GOTO Pipe
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 Device: STATUS @Resource,1;Reg_1
850     PRINT "@Resource assigned to device"
860     PRINT "at interface select code ";Reg_1
870     GOTO Common_exit
880 File: STATUS @Resource,1;Reg_1,Reg_2,Reg_3
890     PRINT "File type          ";Reg_1
900     PRINT "Device selector    ";Reg_2
910     PRINT "Number of sectors ";Reg_3
920     GOTO Common_exit
930 Buffer: STATUS @Resource,1;Reg_1,Reg_2
940     PRINT "Buffer type          ";Reg_1
950     PRINT "Buffer size (bytes) ";Reg_2
960     GOTO Common_exit
970 Pipe: PRINT "HP-UX Pipe"
980 Common_exit: ! Exit point of this routine.
```

BASIC/UX only

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*, you should access the *interface registers* for further information.
- If the name has been assigned to a *file* or *buffer* or *pipe*, you should access the *I/O path registers*.

I/O path names can be assigned to device selectors, files, buffers, and HP-UX pipes. 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 "Data1" ! 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.
```

17

17-6 Registers

```

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     FNEED ! *****

```

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 FN\$C 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 1 = I/O path name assigned to a device 2 = I/O path name assigned to a data file 3 = I/O path name assigned to a buffer 4 = I/O path name assigned to an HP-UX special file (BASIC/UX only)
-------------------	---

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 (not supported for HFS on BASIC/UX)
STATUS Register 3	Number of records
STATUS Register 4	Bytes per record = 256
STATUS Register 5	Current record
STATUS Register 6	Current byte within record
STATUS Register 9	File I/O buffering in use (BASIC/UX only)

17

CONTROL Register 9 Set file I/O buffer (BASIC/UX only)
CONTROL Register 10 Flush I/O buffer (BASIC/DOS only)

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 byte within record
STATUS Register 7 EOF record
CONTROL Register 7 Set EOF record
STATUS Register 8 Byte within EOF record
CONTROL Register 8 Set byte within EOF record
STATUS Register 9 File I/O buffering in use (BASIC/UX only)
CONTROL Register 9 Set file I/O buffer (BASIC/UX only)
CONTROL Register 10 Flush I/O buffer (BASIC/DOS only)

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 byte within record
STATUS Register 7	EOF record
CONTROL Register 7	Set EOF record
STATUS Register 8	Byte within EOF record
CONTROL Register 8	Set byte within EOF record
STATUS Register 9	File I/O buffering in use (BASIC/UX only)
CONTROL Register 9	Set file I/O buffer (BASIC/UX only)
CONTROL Register 10	Flush I/O buffer (BASIC/DOS only)

I/O Path Names Assigned to a DOS File (BASIC/DOS Only)

The following registers are implemented by BASIC/DOS for the DFS file system. For information about DFS file buffering, refer to *Installing and Using HP BASIC/DOS 6.2*.

STATUS Register 1	File type = 4
STATUS Register 2	Device selector of mass storage device
STATUS Register 3	Number of defined records
STATUS Register 4	Defined record length (fixed record length = 1)
STATUS Register 5	Current record
CONTROL Register 5	Set record
STATUS Register 6	Current byte within record
CONTROL Register 6	Set byte within record
17 STATUS Register 7	EOF record
CONTROL Register 7	Set EOF record
STATUS Register 8	Byte within EOF record
CONTROL Register 8	Set byte within EOF record
STATUS Register 9	File I/O buffering in use

17-10 Registers

CONTROL Register 9 Set file I/O buffer
 CONTROL Register 10 Flush I/O buffer

I/O Path Names Assigned to a Buffer

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	TRANS-FER active	TRANS-FER aborted	TRANS-FER error	Device termination	Byte count	Record count	Match character
value=0	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	TRANS- FER active	TRANS- FER aborted	TRANS- FER error	Device termi- nation	Byte count	Record count	0
value=0	value=64	value=32	value=16	value=8	value=4	value=2	value=0

STATUS Register 12

Total number of bytes transferred by last inbound TRANSFER

STATUS Register 13

Total number of bytes transferred by last outbound TRANSFER

I/O Path Names Assigned to HP-UX Special Files (BASIC/UX only)

STATUS Register 1

1 = I/O path is assigned to an HP-UX special file

STATUS Register 2

0 = inbound pipe
1 = outbound pipe
2 = bi-directional pipe

Direct Interface Access

The third level of register access provides *direct* access to interface hardware; this level of access is identical to that possessed by the operating-system firmware. Consequently, these interface-access techniques should *only* be used if you have a *complete* understanding of both the specified register's definition and of the consequences of reading from or writing to these registers. The READIO and WRITEIO interface register definitions and access methods are listed in the *HP BASIC 6.2 Interface Reference* manual.

17

Interrupts and Timeouts

The computer can sense and respond to the occurrence of several types of interrupt events. This chapter describes programming techniques for handling the interface events called “interrupts” and “timeouts” which can initiate program branches. For more details on event-initiated branches, consult the “Program Structure and Flow” chapter and the *HP BASIC Language Reference* descriptions of the keywords described in this chapter.

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 This event occurs when one of the nine “knobs” (rotary pulse generators) of an HP 46085 Control Dial Box is turned. (See the “Communicating with the Operator” chapter of *HP BASIC 6.2 Advanced Programming Techniques* for details.)

ON END	This event occurs when the computer encounters the end of a mass storage file while accessing the file. (See the “Data Storage and Retrieval” chapter for details.)
ON ERROR	This event occurs when a program-execution error is sensed. (See the “Handling Errors” chapter for details.)
ON EXT SIGNAL	This event occurs when an HP-UX signal is sensed (BASIC/UX only). (See the “Trapping HP-UX Signals” section of this chapter for details.)
ON HIL EXT	This event occurs when input from an HP-HIL device is sensed. (See the “HP-HIL Interfaces” chapter of the <i>HP BASIC 6.2 Interface Reference</i> manual for details.)
ON KEY	This event occurs when a currently defined softkey is pressed. (See the “Program Structure and Flow” chapter of this manual or the “Keyboard Interfaces” chapter of the <i>HP BASIC 6.2 Interface Reference</i> manual for details.)
ON KNOB	This event occurs when the “knob” (rotary pulse generator) is turned. (See the “Program Structure and Flow” chapter of this manual and the “Keyboard Interfaces” chapter of the <i>HP BASIC 6.2 Interface Reference</i> for details.)
ON INTR	This event occurs when an interrupt is requested by a device or when an interrupt condition occurs at the interface. (Discussed in the next sections of this chapter.)
ON TIMEOUT	This event occurs when the computer has not detected a handshake response from a device within a specified amount of time. (Discussed in the “Interface Timeouts” section of this chapter.)

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** (**k1** and **k2** on 98203 keyboards) being pressed; the software priorities assigned to these events are 3 and 4, respectively. Run the program and alternately press these softkeys; the branch

18

18-2 Interrupts and Timeouts

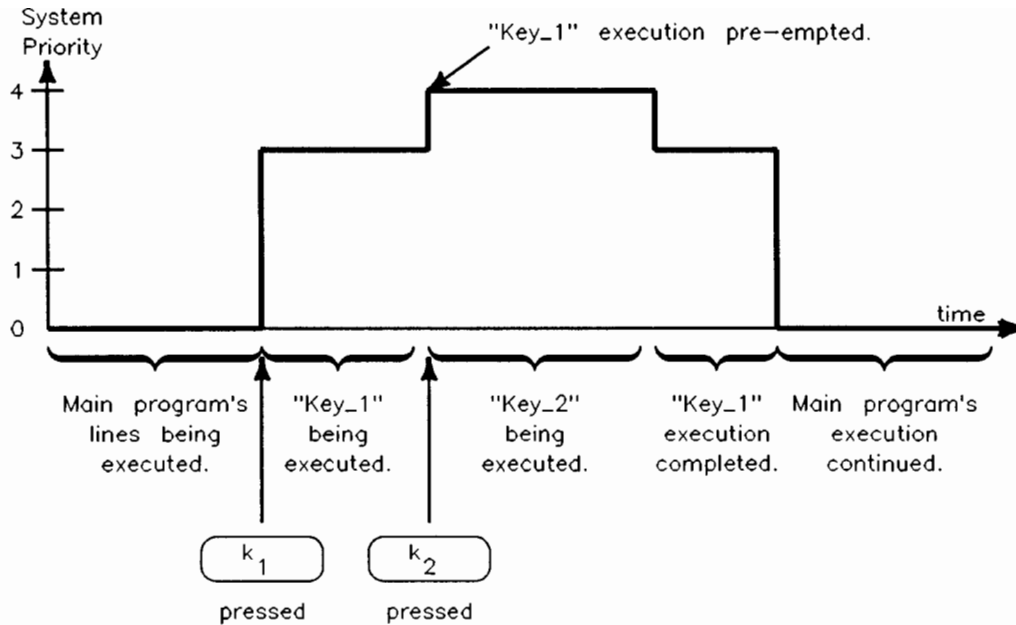
to each key's service routine is initiated by pressing the key. The system priority is "graphed" on the CRT display.

```

150 ON KEY 1,3 CALL Key_0 ! Set up events and
160 ON KEY 2,4 CALL Key_1 ! 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$&"0"
210 !
220 Main: CALL Bar_graph(7,"*") ! Sys. prior. is
230 ! always >= 0.
240 BEEP 100,.1 ! Low tone.
250 FOR Jiffy=1 TO 5000
260 NEXT Jiffy
270 !
280 GOTO Main ! Main loop.
290 !
300 END
310 !
320 SUB Key_1
340 BEEP 300,.1 ! Middle tone.
350 FOR Iota=1 TO 2000
360 NEXT Iota
370 CALL Bar_graph(4," ") ! Erase.
380 SUBEND
390 !
400 SUB Key_2
410 CALL Bar_graph(3,"*") ! Graph priority.
420 BEEP 400,.1 ! High tone.
430 FOR Twinkle=1 TO 2000
440 NEXT Twinkle
450 CALL Bar_graph(3," ") ! Erase.
460 SUBEND
470 !
480 SUB Bar_graph(Line,Char$)
490 CONTROL 1,1;Line ! Locate line.
500 OUTPUT 1;Char$ ! Bar-graph character.
510 SUBEND

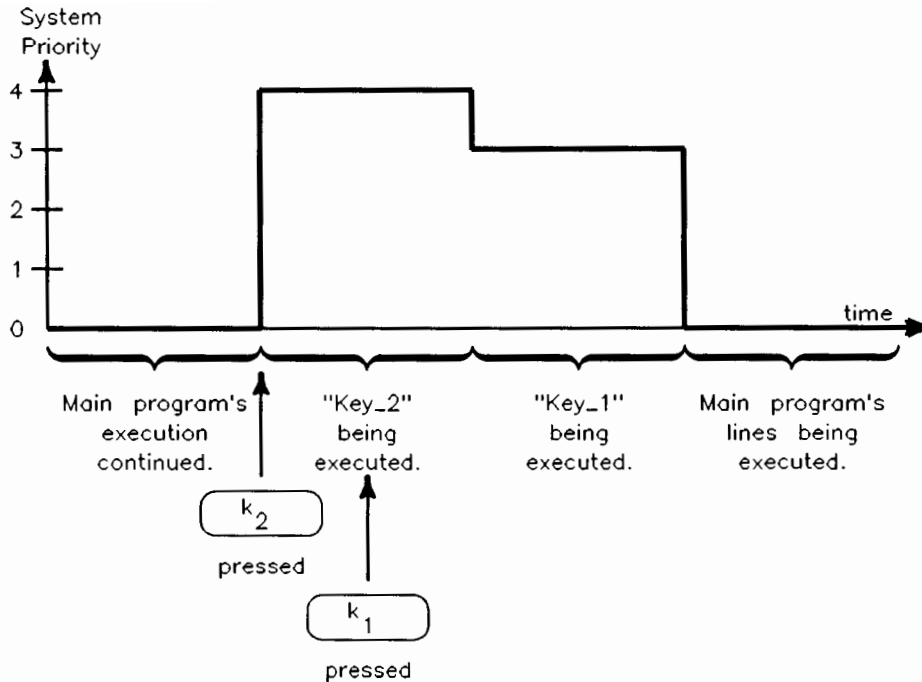
```

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**.



Events with Higher Software Priority Take Precedence

On the other hand, if f_1 is pressed *while* f_2 is being serviced, the computer finishes executing Key_2 *before* executing Key_1. The event of pressing f_1 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. The

“Program Structure and Flow” chapter of this manual and the *HP BASIC Language Reference* fully describe the differences between these types of branches.

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. See the “Subprograms” chapter of this manual or the *HP BASIC Language Reference* for further details.

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
```

Further details of enabling these events are described in the “Interface Interrupts” and “Interface Timeouts” sections of this chapter.

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. (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.)

When all of these conditions have been met, the branch is taken.

Logging and Servicing Events

The preceding events may occur at any time; however, the computer is only “concerned” if these events have been “set up” to initiate a branch. An example of the computer ignoring an event is seen when an undefined softkey is pressed. Since the event has not been set up, the computer beeps. No service routine is executed, even though the computer was “aware” of the event. Thus, only when an event is first set up and then occurs does the computer “service” its occurrence.

Software Priority

The computer first “logs” the occurrence of an event which is set up. (See the section “Hardware Priority” for a description of the process of logging event occurrences.) 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

Software Priority (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 <i>cannot</i> be changed.
17	Effective software priority of ON ERROR; the software priorities of these events <i>cannot</i> 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 preempted (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:

- 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.

18

18-8 Interrupts and Timeouts

```

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.
- 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. (For information on setting hardware priority on optional interfaces, see the interface's installation manual.) *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 Series 200/300 computers. These priorities also apply to the HP Measurement Coprocessor. Only the optional interfaces' hardware priorities can be changed.

Hardware Priorities of Interfaces

Hardware Priority	Interface(s) and Event(s) at This Priority
0	(Quiescent level; no interface is currently interrupting)
1	Built-in Keyboard (KEY and KNOB events)
2	Built-in Disk Drive of 226/236 (END event)
3	Built-in HP-IB or Serial interfaces (INTR and TIMEOUT events)
3-6	Optional Interface Cards (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 *pending service* by the computer. The computer can then return to its former task (unless other events have occurred which have not been logged).

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.

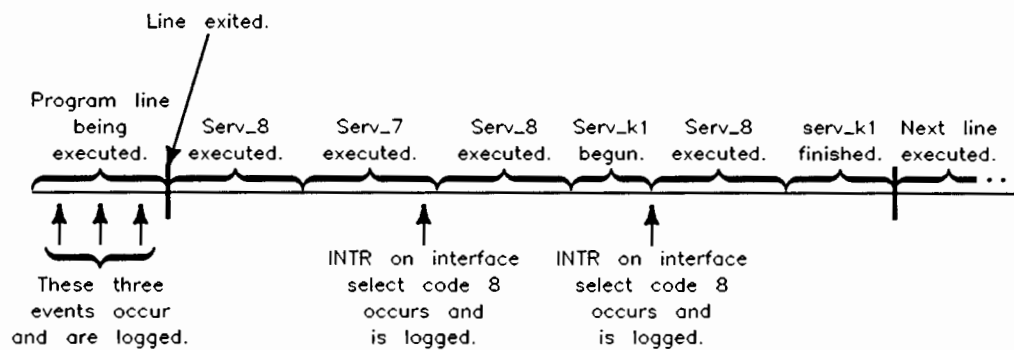
1. Events of higher hardware priority will interrupt the current activity to be logged by the computer.
2. Events which also have higher software priority will interrupt the computer's present activity to be serviced.

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.

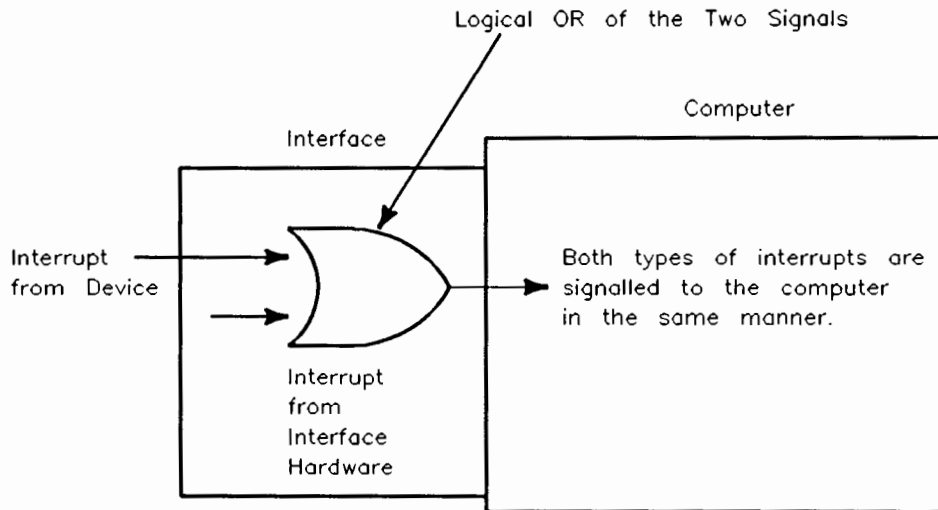


INTR Event Servicing Example

Hardware priority's main function is to keep events of lower hardware 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.



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.

The specific abilities of each interface to detect interrupt conditions and to pass on service requests from devices are described in the *HP BASIC 6.2 Interface Reference* manual.

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.

Example

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.

Enabling and Disabling Events with WRITEIO

This section shows how to use WRITEIO to perform the same functions as ENABLE INTR and DISABLE INTR statements. The examples are shown for the HP 98630 Breadboard Interface, an interface for which no driver is installed (and therefore will not permit ENABLE INTR and DISABLE INTR to be used).

ON INTR and OFF INTR statements may be executed for *any* I/O card plugged into the computer. However, if there is no driver currently loaded for an interface card, all other I/O statements (CONTROL, STATUS, ENABLE INTR, OUTPUT, ASSIGN, etc.) will generate an **Error 163 I/O interface not present** message. Before an interrupt can be generated by a “driverless” interface card, you must emulate the ENABLE INTR statement by using WRITEIO. For example, if an HP 98630 Breadboard card is at select code 17, the following statements set up the service routine “My_card_isr” and enable interrupts for this card:

```
100 ON INTR CALL My_card_isr
110 WRITEIO 17,Mask_reg;Mask_value ! Set the mask.
120 WRITEIO 17,3;128 ! Enable interrupts.
```

The two WRITEIO statements simulate the function of the ENABLE INTR statement.

When the Breadboard card interrupts, BASIC clears bit 7 of WRITEIO register 3 (the interrupt enable bit) and logs the interrupt so that the service routine will be called the next end-of-line (if system priority permits). No other actions are taken during the *hardware* interrupt-logging routine; however, the *software* service routine is free to do whatever you want it to do.

To perform a DISABLE INTR function, execute this statement:

```
300 WRITEIO 17,3;0 ! Disable interrupts.
```

Use this information as required, especially if you wish to use the HP 98630 Breadboard card for customized I/O.

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.

Example

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 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.
.
```

18

18-20 Interrupts and Timeouts

```

.   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 following statements set up this 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.

Example

Set up a timeout to occur after the Serial Interface has not detected a response from the peripheral after 0.200 seconds. Branch to a subroutine called "Serial_down".

```
ON TIMEOUT 9,.2 GOSUB Serial_down
```

Example

Set up a timeout of 0.060 for the interface at select code 8.

```
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 and serial interfaces.

With BASIC/UX, you cannot set up a timeout on an interface when burst I/O mode is in effect.

Timer Resolutions

Timer resolutions are as follows:

- 10 ms for BASIC/WS and BASIC/DOS
- 20 ms for BASIC/UX

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.

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.

No Default Timeout Parameter

There is *no default* timeout time parameter. Thus, if no ON TIMEOUT is executed for a specific interface, the computer will wait *indefinitely* on the device to respond. The only way that the computer can continue executing the program is to use the **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.

Trapping HP-UX Signals from BASIC/UX

The HP-UX system generates signals when certain conditions occurs—such as when it encounters a bad system call (SIGSYS). This section explains how to trap these signals using BASIC/UX end-of-statement branches.

Relevant BASIC/UX Keywords

The following keywords are used when dealing with event-initiated branches that are generated by system signals.

ON EXT SIGNAL	defines an event-initiated branch to be taken when an HP-UX system-generated signal is received.
ENABLE EXT SIGNAL	enables HP-UX signals to initiate branches.



STATUS 33,1;Signal_1 through
STATUS 33,32;Signal_32

Registers 1 through 32 correspond to the 32 HP-UX signals. (See the *HP BASIC Language Reference* description of ON EXT SIGNAL for a complete list of signals and their corresponding register numbers.) These registers may have the following values:

1: indicates that the signal is enabled to initiate a BASIC/UX branch.

0: indicates that the signal is currently disabled from initiating a BASIC/UX branch.

-1: indicates that the signal is not trappable by BASIC/UX or is not supported on HP-UX.

DISABLE EXT SIGNAL

temporarily disables interrupts from HP-UX signals (note that the branch is still defined, though). You can use ENABLE EXT SIGNAL to re-enable the branch.

OFF EXT SIGNAL

Cancels event-initiated branches previously defined by an ON EXT SIGNAL statement. This keyword restores the default actions for the signal.

Example

Here is an example program that shows how to trap HP-UX signals.

```
100 DIM Killstring$[80]
110 ON EXT SIGNAL 15,15 GOTO Leave ! Catch SIGTERM (termination signal).
120 Killstring$="(sleep 5; kill "&SYSTEM$("PROCESS ID")&") &"
130 EXECUTE Killstring$;WAIT OFF ! Send BASIC/UX SIGTERM in 5 seconds.
140 ! Do real work here.
150 Idle: GOTO Idle
160 Leave: PRINT "TIME TO GO"
170 ! Cleanup before exiting.
180 QUIT ! Exit BASIC/UX.
190 END
```

18

I/O Path Attributes

This chapter contains two major topics, both of which involve additional features provided by I/O path names.

- The first topic is that I/O path names can be given attributes which control the way that the system handles the data sent and received through the I/O path. Attributes are available for such purposes as controlling data representations, generating and checking parity, and defining special end-of-line (EOL) sequences.
- The second topic is that one set of I/O statements can be used to access most system resources, including the CRT display, the keyboard, mass storage files, and buffers (instead of using a separate set of BASIC statements to access each class of resources). This second topic, herein called “unified I/O”, may be considered an implicit attribute of I/O path names.

The FORMAT Attributes

All I/O paths 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.

Two FORMAT Attributes Are Available

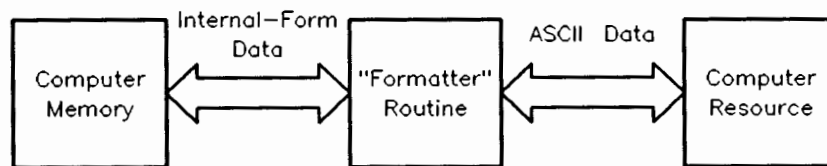
All I/O paths possess one of the two following attributes:

- **FORMAT ON**—means that the data are sent in ASCII representation.
- **FORMAT OFF**—means that the data are sent in BASIC internal representation.

Before getting into how to assign these attributes to I/O paths, let's take a brief look at each one. (Complete descriptions of these data representations are given in the "Interfacing Concepts" section.)

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 diagrams below:

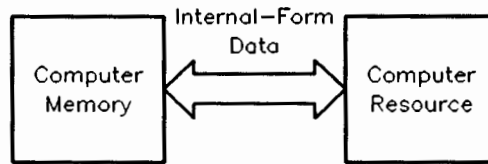


Numeric Data Transformations with FORMAT ON

For more information about the ASCII data format, see the "Interfacing Concepts" section. For details of how items and I/O statements are terminated, see the "Outputting Data" and "Entering Data" chapters.

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 are how each type of data item is represented and sent with **FORMAT OFF**:

- **INTEGER**: two-byte (16-bit), two's complement.
- **REAL**: eight-byte (64-bit) IEEE floating-point standard.
- **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 chapter).
- If either **OUTPUT** or **ENTER** uses an **IMAGE** (such as with **OUTPUT 701 USING "4D.D"**), then the **FORMAT ON** attribute is *automatically* used.

Assigning Default FORMAT Attributes

As discussed in the “Directing Data Flow” section, 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, pipes 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.
```

19-4 I/O Path Attributes

- 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

Several other I/O path attributes may be specified with ASSIGN to:

- append data to an existing file rather than overwrite it (APPEND)
- send and receive data on a byte or word basis (BYTE and WORD)
- perform conversions on a character-by-character basis on inbound and/or outbound data (CONVERT)
- re-define the end-of-line sequence normally sent after the last data item in output operations (EOL)
- check for parity on inbound data, and generate parity on outbound data (PARITY)

It is also possible to direct BASIC 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 APPEND Attribute

The APPEND attribute is useful for I/O paths assigned to files. If you do not specify APPEND when you assign an I/O path name to a file, any data in already in the file is overwritten. If you specify APPEND, new data is appended to any data already in the file.

```
ASSIGN @Outfile TO "Results"           Overwrites the "Results" file
ASSIGN @Outfile TO "Results";APPEND    Appends data to the "Results" file
```

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 the HP 98622 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
```

19-6 I/O Path Attributes

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.

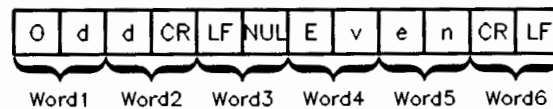
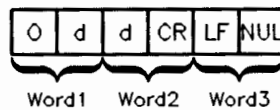
With the **FORMAT OFF** attribute, all data items are represented by an even number of bytes (see the discussion in “The **FORMAT OFF** Attributes” earlier in this chapter for details). Since these representations use an even number of bytes, no pad bytes are necessary.

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.



Characters Sent by **OUTPUT** Statements Shown Above

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

19-8 I/O Path Attributes

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"
```

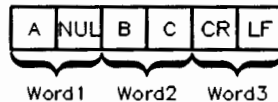
Notice that three ENTER statements were used to enter the data sent by the two preceding OUTPUT statements. This method was used to handle the pad bytes generated by the OUTPUT statement. If two ENTER statements would have been used, the pad byte sent after the second "Odd" and EOL sequence would have 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.

19-10 I/O Path Attributes

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

19-12 I/O Path Attributes

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$=Conv_string$&"qQrRsStTuUvVwWxYyZz"
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. You can also define your own special EOL sequences that include sending special characters, sending an END indication, and delaying a specified amount of time after sending the last EOL character.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. Here is an example that changes the EOL sequence to a single line-feed character. It is useful, for instance, in sending data to files that you will be later editing with the HP-UX vi editor (or sending to HP-UX pipes):

```
ASSIGN @File TO "file_one";EOL CHR$(10)
```

The characters following the secondary keyword EOL are the EOL characters. Any character in the range CHR\$(0) through CHR\$(255) may be included in the string expression that defines the EOL characters; however, the length of the sequence is limited to eight characters or less. 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. The individual chapter that describes programming each interface (in the *HP BASIC 6.2 Interface Reference* manual) further describes each interface's END indication (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. See the chapters that describe interrupts in general and interrupts for the specific interface.

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

of attributes directs the system to place 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
```

19-18 I/O Path Attributes

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.

Transfers and Buffered I/O

This chapter discusses data transfer techniques. 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

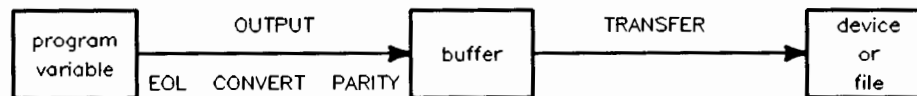
Every transfer will use a buffer as either its source or its destination. From the buffer’s point of view, there are two types of transfers.

An **inbound** transfer moves data from a device or file into the buffer.



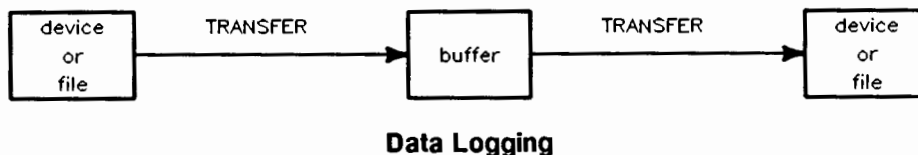
Inbound Transfer

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.



Supported Transfer Sources and Destinations

TRANSFER operations are allowed only for certain types of interfaces, files, and pipes.

Note



A transfer *cannot* involve a CRT display, a keyboard, an ASCII file, a BCD interface, or a DC600 cartridge-tape drive.

For the HP Measurement Coprocessor, a transfer *cannot* involve the PC serial ports (COM1 and COM2) or PC parallel port (LPT1).

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.

Further restrictions are listed in the "TRANSFER Restrictions" section later in 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 Measurement Coprocessor.

Supported Interfaces

	HP 9000 Series 200/300	HP Measurement Coprocessor
HP-IB	Built-in and HP 98624	Built-in, HP 82335, HP 82990
GPIO	HP 98622	HP 82306
Serial	Built-in, HP 98626, HP 98644	Not supported
Datacomm	HP 98628 HP 98642 (BASIC/UX only)	Not supported
Parallel	Built-in	Not supported
SCSI	Built-in and HP 98658	Not supported

File Types

BDAT and HP-UX files (and DOS files for the HP Measurement Coprocessor) are supported sources and destinations for the TRANSFER statement. ASCII files are not supported as sources or destinations.

Pipes (BASIC/UX only)

Named and unnamed pipes are supported as TRANSFER sources and destinations. Refer to the *HP BASIC 6.2 Advanced Programming Techniques* manual for further information.

Transfer Examples

20

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 also be accessed by its variable name (for instance, by using OUTPUT or assigning the variable). However, an unnamed buffer can only be accessed 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)
```

20-6 Transfers and Buffered I/O

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.

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 only be accessed 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. This allows a buffer to be larger than the maximum length of 32 767 bytes for 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.

20 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 assigned to a device
	2 = I/O path assigned to a data file
	3 = I/O path assigned to a buffer
	4 = I/O path assigned to an HP-UX special file (BASIC/UX only)

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 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 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.

20-8 Transfers and Buffered I/O

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.

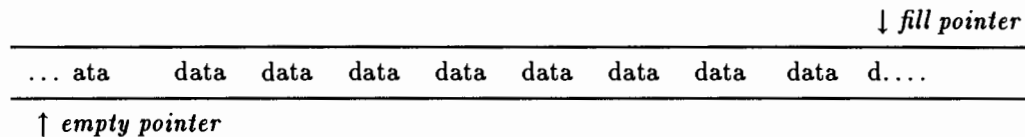
- The **buffer fill pointer** points to the next byte of the buffer which can accept data.
- The **empty pointer** points to the next byte of data which can be read from the buffer.
- The **byte count** shows the number of bytes currently in the buffer (usually equal to fill pointer – empty pointer).

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.

TRANSFER @Buffer to @File *inbound transfer*



TRANSFER @Buffer to @File *outbound transfer*

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) to become equal to the buffer capacity (Register 2). 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.

Both full and empty buffers have 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, buffer or pipe found

20-10 Transfers and Buffered I/O

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 via Variable Names

If you plan to transfer data through a buffer without using the I/O path name (such as by using the string variable'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 for BASIC/WS

The transfer methods available for BASIC/WS are:

- DMA (direct memory access)
- FHS (fast handshake)
- INT (interrupt)

Descriptions of each method and how BASIC chooses one for each TRANSFER are covered in the section “Transfer Methods and Rates”.

Transfer Methods for BASIC/DOS

BASIC/DOS supports the same transfer methods as BASIC/WS, except that only the HP 82324 High-Performance Measurement Coprocessor supports DMA transfers, not the HP 82300 Measurement Coprocessor.

Transfer Methods for BASIC/UX

The transfer methods available for BASIC/UX are:

- DMA (direct memory access).
- Burst (non-buffered I/O, with the interface locked to the BASIC/UX process).
- Buffered I/O (the default mode when DMA is not available and burst is not specifically enabled).

Further descriptions of each method and how the system chooses one for each TRANSFER are covered in the subsequent section called “Transfer Methods for BASIC/UX.”

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

The ON EOT (End Of Transfer) statement allows you to define a branch to be taken upon the completion of a transfer. When the data being transferred has been divided into records, the ON EOR (End Of Recordr) 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 (**I/O** or **Transfer**) is displayed in the lower-right corner of the CRT instead of the “Pause” indicator (**-** or **Paused**).

Choosing Transfer Parameters

For a standard inbound transfer, data from the device (or file) is placed in the buffer and the TRANSFER is deactivated when the buffer is full. For an outbound transfer, all data is removed from the buffer and the TRANSFER is deactivated when the buffer is empty.

Continuing Transfers Indefinitely

To allow a TRANSFER to continue indefinitely, the CONT parameter can be specified.

```
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 completed 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
```

Continuous Non-Overlapped Transfers

When the WAIT parameter is specified, the program statement following the TRANSFER will not be executed until the transfer has completed.

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 it 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	Device/File Response
No END	Terminate prematurely. Bit 3 of Register 10 is set. Error 59 waiting.
END	Terminate normally. Bit 3 of Register 10 is set.
EOR(END)	<i>Files</i> Finish current record. ON EOR triggered. Start new record. <i>Devices</i> Terminate normally. Bit 3 of Register 10 is set.
END, EOR(END)	Terminate normally. 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. 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.

20-18 Transfers and Buffered I/O

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.

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.

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

```

20-22 Transfers and Buffered I/O

```

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

```

20-24 Transfers and Buffered I/O

```

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.

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. A sensible choice is:

```
WAIT FOR EOT @Non_buff
```

The system also will not process an ON EOT (or ON EOR) branch defined for the transfer if the

```
ASSIGN @Path TO *
```

statement begins execution before the ON event has been logged. Again, WAIT FOR EOT @Non_buff prior to the ASSIGN @Path TO * will ensure that these events will be logged and serviced.

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 Which 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
	OFF EOR @Dev
	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

ASSIGN @Buf TO *
 ASSIGN @Buf TO BUFFER[bytes]
 ASSIGN @Buf TO BUFFER B\$
 ASSIGN @Dev
 ASSIGN @Dev; (*new attributes*)

END
 SUBEXIT
 SUBEND
 SCRATCH C
 SCRATCH
 LOAD "PROG"
 GET "PROG"
 STOP
 ABORT
 SEND

Device in Use

ASSIGN @Dev TO *
 ASSIGN @Dev
 ASSIGN @Dev; (*new attributes*)
 WAIT FOR EOT @Dev
 OUTPUT @Dev
 ENTER @Dev
 TRANSFER @Buf TO @Dev
 TRANSFER @Dev TO @Buf

END
 SUBEXIT
 SUBEND
 SCRATCH C
 SCRATCH
 LOAD "PROG"
 GET "PROG"
 STOP
 CONTROL @Dev

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
```

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 files, the buffer size should be a multiple of 256 or 1024 bytes (the size of a sector on the disk). Note that BASIC does not support 512-byte sectors. 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.

20-30 Transfers and Buffered I/O

Internal Disk Drives of Models 226 and 236 Computers

20

While a TRANSFER can be assigned to the internal disk drives in the Model 226 and Model 236, no noticeable increase in speed (compared to OUTPUT or ENTER) will result. Transfers to and from external mass storage (except the 9885) will show an increase in speed, especially if a DMA card is present.

Overlapped Transfers and Disk Drives

Some of the disks are capable of overlapped operation. This means that other processing can occur while a non-continuous TRANSFER to or from the disk is taking place. In other words, the program can execute other statements before the transfer has completed. Overlapped disks include:

- CS80 disks (such as the HP 9153)
- SS80 disks (such as the HP 9122)
- “Amigo” disks (such as the HP 9895 and HP 82901)
- SCSI disks (such as the HP C1701A)

Disks which are not capable of overlapped operation are called serial disks. When executing a non-continuous TRANSFER to or from a serial disk, the program will not leave the TRANSFER statement until it completes. Serial disks include the internal disks (of Models 226 and 236 computers) and the HP 9885 8-inch flexible disk drive. With files on HFS and SRM volumes, the TRANSFER statement runs in overlapped mode until the BASIC system encounters a statement that accesses the same volume (such as CAT or ASSIGN); at such times, the BASIC system performs an implicit WAIT FOR EOT.

The following example illustrates the difference between a serial disk and an overlapped disk.

```

10  OPTION BASE 1
20  INTEGER B(128,10)           ! A 10-sector buffer
30  LINPUT "Enter msus:",Msus$
40  CREATE BDAT "bdat"&Msus$,10
50  ASSIGN @File TO "bdat"&Msus$
60  ASSIGN @Buffer TO BUFFER [2560];FORMAT OFF
70  OUTPUT @Buffer;B(*)         ! Fill @Buffer's with 10 sectors
80  ON EOT @File GOTO Serial_eot ! Branch taken if TRANSFER is serial
90  TRANSFER @Buffer TO @File
100 ON EOT @File GOTO Overlapped_eot! Branch taken if TRANSFER is overlapped
110 LOOP
120   I=I+1
130   PRINT I,"OVERLAPPED"
140 END LOOP
150 Serial_eot: !
160 PRINT "SERIAL"
170 Overlapped_eot: !
180 ASSIGN @File TO *
190 PURGE "bdat"&Msus$
200 END

```

If this program is used with a serial disk, the program stays in the TRANSFER statement until the transfer is complete. Upon completion of the transfer, the ON EOT branch to Serial_eot is taken.

If this program is used with an overlapped disk, the TRANSFER statement begins the transfer, but the program executes the next statement before the transfer completes. In this program, the next statement changes the ON EOT branch. During the transfer, a count and the word "OVERLAPPED" are printed. When the transfer is complete, the ON EOT branch to Overlapped_eot is taken.

If the CONT parameter is specified for a TRANSFER with a serial disk, the transfer may appear overlapped in certain cases because the program executes any statements which follow the TRANSFER statement before the transfer terminates. Here is what really happens in this case. The transfer proceeds until the buffer is full (for inbound transfers) or empty (for outbound transfers). The transfer is then suspended because CONT was specified. The TRANSFER statement is exited and the next statement is executed. The transfer will remain suspended until the continuous mode is terminated or until

20-32 Transfers and Buffered I/O

the buffer is filled (for inbound transfers) or until the buffer is emptied (for outbound transfers). If there is a second TRANSFER active for the buffer, an EOR or EOT condition for the second TRANSFER can also wake up the suspended TRANSFER.

In contrast to serial disks, overlapped disks would allow the statement following the TRANSFER to execute before the buffer was full or empty.

The following program illustrates a transfer to a serial device which appears overlapped.

```

10  OPTION BASE 1
20  INTEGER B(128,10)           ! A 10-sector buffer
30  LINPUT "Enter Overlapped msus:",Overlapped$
40  CREATE BDAT "bdat"&Overlapped$,10
50  LINPUT "Enter Serial msus:",Serial$
60  CREATE BDAT "bdat"&Serial$,10
70  ASSIGN @Overlapped TO "bdat"&Overlapped$
80  OUTPUT @Overlapped;B(*)
90  RESET @Overlapped           ! Position to beginning
100 ASSIGN @Buffer TO BUFFER [512];FORMAT OFF
110 ASSIGN @Serial TO "bdat"&Serial$
120 ON EOT @Overlapped GOTO Eof
130 TRANSFER @Overlapped TO @Buffer;END,CONT
140 TRANSFER @Buffer TO @Serial;CONT
150 LOOP
160   I=I+1
170   PRINT I,"OVERLAPPED"
180 END LOOP
190 Eof: !
200 CONTROL @Buffer,9;0
210 ASSIGN @Overlapped TO *
220 PURGE "bdat"&Overlapped$
230 ASSIGN @Serial TO *
240 PURGE "bdat"&Serial$
250 END

```

In this example, an overlapped disk is used to fill the buffer while a serial disk empties the buffer. Any overlapped device could have been used. After both TRANSFER statements are executed, the program prints the count and the word "OVERLAPPED" while reading from one disk and writing to the other disk. The inbound transfer is terminated when it encounters the end of the file. The outbound transfer is terminated when the CONTROL statement cancels the CONT mode.

20 Transfer Methods and Rates

BASIC chooses the *fastest possible* transfer method when executing a TRANSFER (you cannot explicitly choose the method).

Available Methods

There are several types of transfers available to the BASIC/WS system.

- DMA (direct memory access)
- FHS (fast handshake)
- INT (interrupt)

BASIC/UX also supports busrt and buffered I/O. The remainder of this section explains BASIC/WS TRANSFER methods. For more information about BASIC/UX TRANSFER methods, refer to the subsequent section of this manual, "Using TRANSFER in BASIC/UX".

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, GPIO, or HP Parallel 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

20

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 μ s for another byte. If the device is fast enough to accept or generate another byte each 20 μ s, 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.

Approximate Transfer Rates for Devices

The following table shows the *approximate* transfer rates of various devices. Note that these rates will vary depending on what system you are using.

Device Transfer Rates

Device	Burst Interrupt	Fast Handshake	DMA	Burst DMA
HP-IB (98624 and built-in)				
inbound	55K	130K	350K	—
outbound (bytes/second)	75K	120K	290K	—
GPIO (98622)				
inbound	65K	115K	540K	930K
outbound (transfers/second)	75K	115K	525K	1050K
Serial (98626, 98644, and built-in)	19 200 Baud ¹	—	—	—
Datacomm (98628)	19 200 Baud	—	—	—

¹ Note that the maximum rate for *inbound* transfers through a Serial interface is generally *much* lower than this for two reasons: TRANSFER does not use the handshake lines, and there is only a 1-character hardware buffer on Serial cards.

Using TRANSFER in BASIC/UX

In BASIC/UX, the TRANSFER statement is handled by executing a separate process, called `rmbxfr`. This process handles the overlapped I/O concurrent with BASIC execution. The `rmbxfr` process is started when the TRANSFER statement is executed, and it runs until the TRANSFER completes, or an ABORTIO statement or the `(Reset)` key kills it. There is an `rmbxfr` process running for each TRANSFER currently executing.

Locking an Interface During a TRANSFER

If you want to reserve an HP-IB or GPIO interface for exclusive use, you can lock it to a process with this statement:

```
CONTROL Select_code,255;1
```

Note that this is NOT considered “good citizenship” on a multi-user HP-UX system, but it is okay in a single-user system. See the section called “Using Interfaces in the HP-UX Environment” found in the “Directing Data Flow” chapter.

If the interface is locked by BASIC/UX, it will be unlocked before the TRANSFER process `rmbxfr` is executed, and then re-locked inside `rmbxfr`. Once the TRANSFER terminates, BASIC/UX re-locks the interface until explicitly unlocked by this statement:

```
CONTROL Select_code,255;0
```

Using the Burst I/O Mode During a TRANSFER

It is possible to improve TRANSFER performance for HP-IB and GPIO interfaces by using Burst I/O Mode to map the interface into your “user address space” with this statement:

```
CONTROL Select_code,255;3
```

See the section called “Using Interfaces in the HP-UX Environment” found in the “Directing Data Flow” chapter for more information. Note that it is also repeated in the chapters “The HP-IB Interface” and “The GPIO Interface” in the *HP BASIC 6.2 Interface Reference* manual.

Transfer Methods for BASIC/UX

There are three types of transfer methods available to the BASIC/UX system:

- Burst I/O
- DMA (direct memory access)
- Buffered I/O

If Burst I/O mode is enabled as described in the previous section, it will be used for the duration of the TRANSFER.

If Burst I/O mode is not in effect, the HP-UX system will attempt to use DMA mode whenever it can. The use of DMA is dependent on whether or not a DMA channel is available, since BASIC/UX does not reserve one.

If a DMA data transfer is not possible, then buffered I/O will be used. Since the TRANSFER statement breaks the data up into segments for processing, it is possible that DMA may be used for some segments of the TRANSFER and

not for others. This will depend on the demand for DMA channels by other activity in the system.

Timing Issues

Due to the way the `rmbxfr` process communicates with the BASIC/UX process, there may be some slight delays in passing information from `rmbxfr` back to BASIC/UX. This occurs because these two processes must share the SPU with other processes in the system. The SPU is allocated to each process for a brief period of time, called a “time slice.” When this time expires, that process loses its turn, and is re-scheduled to continue execution at a later time.

The time slice scheduling may cause some effects with the handling of asynchronous events associated with a TRANSFER. For example, if the `rmbxfr` process is able to trigger two ON EORs during its time slice, only one of these will be serviced during the next time slice of BASIC/UX. This occurs because only one EOR can be logged at a time for a given I/O path name. Since the first event has not been serviced yet, the second will not be logged.

All TRANSFERs Use Only One BUFFER

Every TRANSFER statement must have one I/O path assigned to a buffer and one I/O path assigned to a device (or file).

Devices and Files Not Supported for TRANSFER

TRANSFERs are *not permitted* with:

- The CRT or keyboard.
- LIF and SRM files (for BASIC/UX only).
- ASCII type files.
- The tape backup on CS80 disk drives.
- Interfaces with HP-UX swap devices or mounted file systems (for BASIC/UX only).
- The HP 98623 BCD interface card.
- A PC serial port (COM1 or COM2) or PC parallel port (LPT1) for the HP Measurement Coprocessor.

In addition, TRANSFERs to or from a mass storage device with hierarchical directories (HFS, DFS, SRM, or SRM/UX) will not operate in overlapped mode.

Note

BASIC/UX *does not* allow the TRANSFER statement to be used with just an HP-IB interface select code number when the HP-IB interface associated with that select code is the Active Controller. You must include a primary bus address with the select code. For example:

```
TRANSFER @Device TO @Buffer
```

where @Device is the source name associated with HP-IB device selector 702.

20 Number of Simultaneous TRANSFERS

A buffer can only have 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 with the buffer in the same direction must wait until the completion of the current operation. A second I/O operation with the buffer in the opposite direction does not have to wait.

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.

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.

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.

20-40 Transfers and Buffered I/O

For the Datacomm interface, all interrupt conditions are triggered even if a transfer is in progress.

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.

Premature Termination

When an overlapped TRANSFER is prematurely terminated because of an abortive interrupt condition, the following error is logged in the non-buffer I/O path name associated with the given TRANSFER. The error will then be reported the next time the I/O path name is referenced.

```
ERROR 167 I/O interface status error
```

It is also possible to get:

```
ERROR 611 Premature TRANSFER termination
```

if something happens to the HP-UX child process used by the TRANSFER (BASIC/UX only).

Note that if an ON INTR condition is triggered during a transfer, the ON INTR service routine will be executed at the next end-of-line. However, if a TRANSFER is using the interface specified in an ENABLE INTR statement, the ENABLE INTR statement will wait for the transfer to complete. This means that only one interrupt condition can be triggered during a TRANSFER since the interface's interrupts cannot be re-enabled until completion of the transfer.

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 *HP BASIC Language Reference* 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.

20

Buffer Status and Control Registers

STATUS Register 0 0 = Invalid I/O path name
 1 = I/O path assigned to a device
 2 = I/O path assigned to a data file
 3 = I/O path assigned to a buffer
 4 = I/O path assigned to an HP-UX special file
 (BASIC/UX only)

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=0	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=0	value=64	value=32	value=16	value=8	value=4	value=2	value=0

- STATUS Register 12 Total number of bytes transferred by last inbound TRANSFER
- STATUS Register 13 Total number of bytes transferred by last outbound TRANSFER

Interface Overview

This chapter provides an overview of the interfaces most commonly used for instrument control — the HP-IB, RS-232 serial, and GPIO interfaces. Specific programming techniques are described for each of these interfaces. These programming techniques apply to the HP Measurement Coprocessor (BASIC/DOS) as well as the HP 9000 Series 200/300 computers (BASIC/WS and BASIC/UX). For a more complete discussion of these interfaces, and of the other available interfaces, refer to the *HP BASIC 6.2 Interface Reference* manual.

Note

The *HP BASIC 6.2 Interface Reference* manual provides a complete discussion of the available Series 200/300 interfaces at a register level. The READIO and WRITEIO registers as well as the STATUS and CONTROL registers are described in detail. Also, the *HP BASIC 6.2 Language Reference* manual provides a summary of the STATUS and CONTROL registers for each interface.

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 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

21

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 the device

21-2 Interface Overview

selector 722. For the device on interface select code 10 with primary address 2, use the 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.

21

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.

21

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

21-4 Interface Overview

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:

21

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$(*)
```

21-6 Interface Overview

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
```

21-8 Interface Overview

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
```

21-10 Interface Overview

```
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.

21

```
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.

21-12 Interface Overview

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.

21

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

21

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Active Controller	0	Primary address of HP-IB interface				
value=128	value=64	value=0	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).

21

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.

21-16 Interface Overview

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.

21

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:

21

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Active Controller	Parallel poll configuration 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	Unrecognized universal command	Secondary command while addressed	Clear received	Unrecognized 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").

21-18 Interface Overview

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.)

21 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

21-20 Interface Overview

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.

21

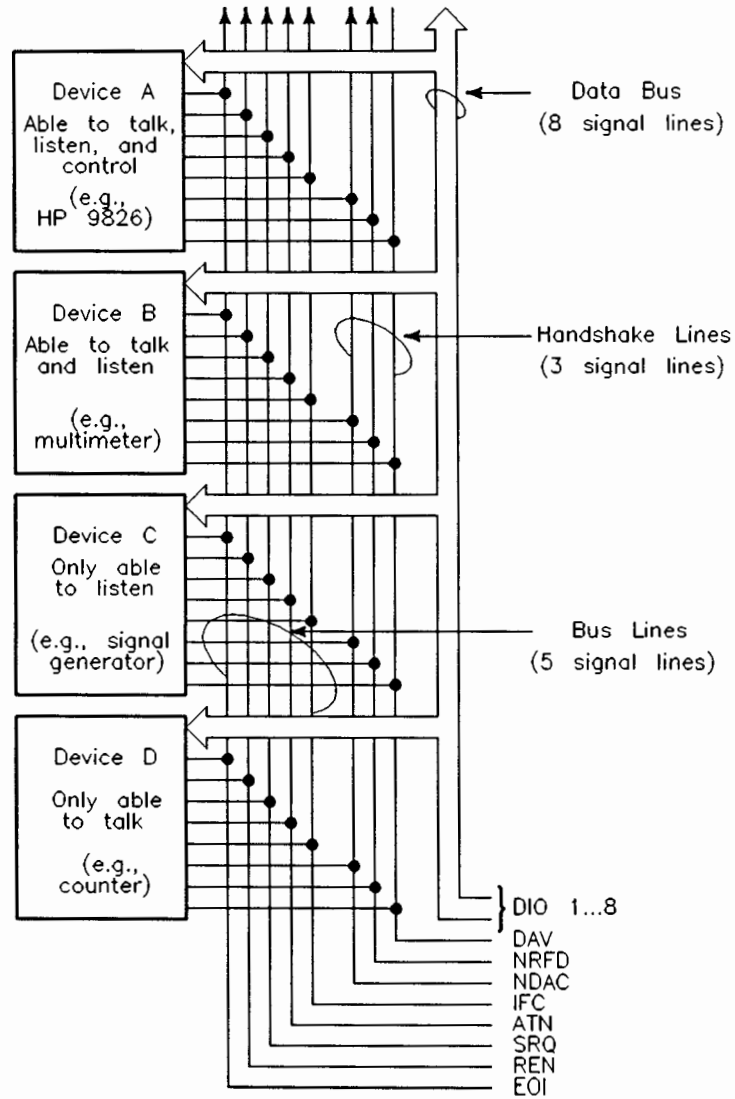
The following statement sets up a response with SRQ and bits 1 and 0 set to “1”.

```
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

21

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.

21-24 Interface Overview

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.

21

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 *HP BASIC 6.2 Language Reference* manual.

21

References

For further information, about the HP-IB (IEEE-488) interface you may want to refer to the following sources:

- *HP BASIC 6.2 Interface Reference*.
- *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 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

21-26 Interface Overview

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.

21

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.

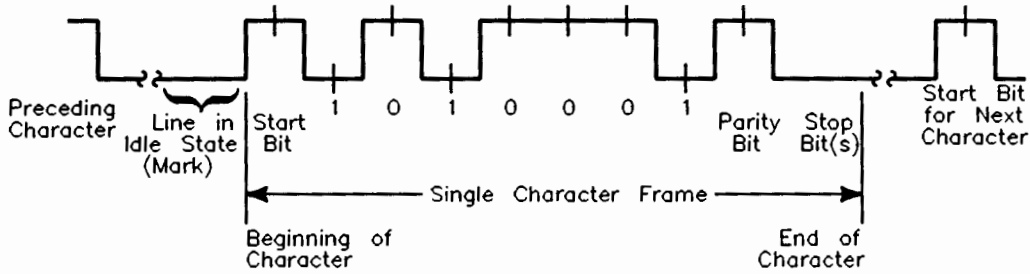
Character Format

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:

21



Asynchronous Format

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 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. 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.
- 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:

- 21
- 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 the *HP BASIC 6.2 Language Reference* manual.

21-30 Interface Overview

Note

The HP Measurement Coprocessor implements the serial interface registers for the PC serial ports (COM1 and COM2). However, serial interface interrupts and serial TRANSFER operations are *not* supported for COM1 and COM2 with the measurement coprocessor.

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.

21-32 Interface Overview

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:

Character Format and Parity Settings

Handshake (Bits ² 7&6)	Parity Sense ¹ (Bits ² 5&4)	Par. Enable (Bit ² 3)	Stop Bits (Bit ² 2)	Char. Length (Bits ² 1&0)
00 no-op	00 ODD parity	0 Disabled	0 1 stop bit	00 5 bits/char
01 Xon/Xoff	01 EVEN parity	1 Enabled	1 2 stop bits	01 6 bits/char
Bidirectional	10 Unsupported			10 7 bits/char
10 Unsupported	11 Unsupported			11 8 bits/char
11 Handshake Disabled				

¹ Parity sense valid only if parity is enabled (bit 3=1). If parity is disabled, parity sense is meaningless.

² These bits correspond to equivalent switch settings on the HP 98626 and HP 98644 serial interface cards. A 1 is the same as set.

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:

Data Transfers

21

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

21-34 Interface Overview

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). 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).

21-36 Interface Overview

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 "GPIOCHECK.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

21

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 handshake 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

21

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.

21-40 Interface Overview

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.

21

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$

String Entered = ruok?
```

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
```

21

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.

Interface Overview 21-43

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 Ok	EIR line low	STI1 line low	STI0 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.

21

Index

A

- Aborting graphics dumps, 9-3
- ABORTIO statement, 20-21
- ABORT statement, 21-7, 21-11
- ABS function, 3-13
- Access of Directories, Extended, 7-18
- ACS function, 3-15
- ACSH function, 3-15
- active controller
 - description, 21-4
- Actual values, 6-7
- Adding Items to a Sorted List, 5-19
- Additional Interface Functions, 14-5
- Address, primary, 14-24
- Allocate memory, Dynamically, 3-4
- ALLOCATE statement, 3-4, 4-2, 4-6, 4-22, 13-3
- Alpha
 - Bit-mapped, 8-64
 - Display-Enable mask, 8-60
 - Scrolling, enable and disable, 8-64
 - Separate from graphics, 8-4
 - Write-Enable mask, 8-60
- Alpha/Graphics interaction, 8-61
- Alpha mask, 8-60
- Alternate CRT Characters, 5-27
- Angular direction (for labels), 8-24
- Animation, 8-64
- Anisotropic scaling, 8-7
- Anisotropic scaling, 8-55
- Anticipating Operator Errors, 12-2
- Appearance of Output, 10-14
- Arbitrary Exit Points, 2-19
- Area
 - soft clip, 8-38
- AREA COLOR statement, 8-54, 8-56, 8-65, 8-69
- AREA INTENSITY statement, 8-48, 8-54, 8-56, 8-65, 8-69
- AREA PEN statement, 8-54, 8-56, 8-65, 8-69
- Area soft clip, 8-9, 8-31
- Argument, 3-22
- Arithmetic Functions, 3-13
- Arithmetic Operators, 3-9
- Array, Copying a Subarray into an, 4-17
- Array, Dynamically Allocated, 4-5
- Array Element, Assigning an Individual, 4-7
- Array, four-dimensional, 4-5
- Array Functions, 3-14
- Array into a Subarray, Copying an, 4-16
- Array into Itself, Copying a Portion of an, 4-19
- Array, Planes of a Three-Dimensional REAL, 4-3
- Array, Printing an Entire, 4-11
- Array, Reordering an, 5-21
- Arrays, Copying Entire Arrays into Other, 4-9
- Arrays, Extracting Single Values From, 4-7
- Arrays, Filling, 4-8

Arrays for Display, Examples of Formatting, 4-11
 Arrays, Passing Entire, 4-13
 Arrays, Printing, 4-11
 Arrays, Redimensioning, 4-21
 Arrays, Searching String, 5-22
 Arrays, Some Examples of, 4-3
 Arrays, Storage and Retrieval of, 7-5
 Arrays, String, 5-3
 Array the Same Value, Assigning Every Element in an, 4-8
 Array, Two-Dimensional REAL Array, 4-4
 Array, Using the READ Statement to Fill an Entire, 4-8
 ASCII Character Set, The, 5-25
 ASCII file, 7-14
 ASCII files, 7-8
 ASCII Representation of Integers, 14-10
 ASCII Representation of Real Numbers, 14-12
 ASN function, 3-15
 ASNH function, 3-15
 Aspect ratio, 8-16, 8-23
 Assigning an Individual Array Element, 4-7
 Assigning Every Element in an Array the Same Value, 4-8
 Assigning Variables, 3-4
 Assignment Surprise, Delayed, 3-11
 ASSIGN statement, 7-10, 7-12, 7-14, 13-3, 14-26, 14-28, 19-4, 20-7
 ASSIGN Statements, Determining the Outcome of, 19-18
 asynchronous I/O, 21-26
 ATN function, 3-15
 ATNH function, 3-15
 attention (ATN) line, 21-4, 21-24
 Attribute, BYTE, 19-6
 Attribute control, 14-31
 Attributes, Additional, 19-5

Attributes, Assigning, 7-13
 Attributes, Changing Buffer, 20-42
 Attributes, FORMAT, 19-1
 Attributes, I/O Path, 19-1
 Attributes, Restoring the Default, 19-5
 Attribute, WORD, 19-6
 Automatic redimensioning, 4-10
 AXES statement, 8-13

B

Backplane, computer, 14-3
 Base Conversion Functions, 3-23
 BASE function, 3-14, 4-6
 BASIC Clock, 11-1
 BASIC Programs, Trapping Errors with, 12-5
 Battery-Backed Real-Time Clock, 11-3
 baud rate
 serial I/O, 21-33
 BDAT file, 7-10
 BDAT files, 7-8, 7-14, 9-10
 BINAND function, 3-16
 Binary, CLOCK, 11-1
 Binary Functions, 3-16
 Binary images, 15-23
 Binary Images, 16-21
 Binary specifier, 15-23
 BINCMP function, 3-16
 BINCMP statement, 9-6
 BINEOR function, 3-16
 BIN files
 GRAPH, 8-1
 GRAPHX, 8-1
 MAT, 8-16
 BINIOR function, 3-16
 BIT function, 3-16
 Bit-mapped alpha, 8-64
 Bit-mapped displays, 8-59
 Bits and Bytes, 14-6
 Bold labels, 8-28
 Boundary Conditions, 12-2

- Branching on Clock Events, 11-14
- Branching Restrictions, 11-18
- Branch, initiating a, 18-5
- Branch Type, Choosing a, 12-5
- breaks
 - serial I/O, 21-29
- Buffer Attributes, Changing, 20-42
- Buffered I/O, 20-37
- Buffer Life Time, 20-8
- Buffer, Named, 20-6
- Buffer Pointers, 20-9
- Buffers, A Closer Look at, 20-6
- Buffers and Transfers, Overview of, 20-2
- Buffers, Creating Named, 20-6
- Buffer Size Register, 20-8
- BUFFER statement, 19-4, 20-6
- Buffer Status and Control Registers, 20-43
- Buffers, Types of, 20-6
- Buffer-Type Registers, 20-8
- Buffer, Unnamed, 20-6
- Burst Interrupt Mode, 20-35
- Burst I/O mode, 20-37
- Burst I/O mode during a TRANSFER, 20-37
- Burst transfers, 20-13
- Bus, 14-3
- BYTE attribute, 19-6, 19-11
- BYTE Attribute, 19-6
- Byte count, 20-9
- C**
- Calling Subprograms, 13-4
- Calling Subprograms from the Keyboard, 6-20
- CALL statement, 2-6, 6-3-4, 6-16-18, 6-20, 6-24
- Case conversion, 5-16
- CASE ELSE statement, 2-14
- CASE statement, 2-14, 5-24
- Catalog Header, Suppressing the, 7-21
- Cataloging Selected Files, 7-22
- Cataloging to a String Array, 7-19
- CAT statement, 7-18
- CAUSE ERROR statement, 12-14
- Ceiling of a number, 8-6
- Cell, character, 8-19
- Chaining Programs, 2-30
- Chapter Previews, 1-3
- Character aspect ratio, 8-23
- Character cell, 8-19
- character length
 - serial I/O, 21-33
- Characters, Control, 10-4
- Characters, Converting, 19-11
- Characters, Display Enhancement, 5-26
- Character Set, CRT, 5-13
- Character set selection, 9-9
- Character Set, The Extended, 5-26
- Characters, Finding "Missing", 5-28
- Characters, Ignoring, 16-20
- Character specifier, 15-21
- Characters, Representing, 14-8
- Characters, user-defined, 8-56
- CharCell program, 8-21
- CHR\$ string function, 5-13, 5-29
- CLEAR ERROR statement, 12-16
- Clearing Error Conditions, 12-16
- CLEAR statement, 21-7, 21-10
- CLIP OFF statement, 8-9, 8-31
- Clipping
 - Hard, 8-9, 8-38
 - Soft, 8-9, 8-38
- CLIP statement, 8-38
- Clock, BASIC, 11-1
- Clock, battery-powered, 11-1
- CLOCK binary, 11-1
- Clock Events, Branching on, 11-14
- Clock Functions and Example Programs, Using, 11-11
- Clock, non-volatile, 11-1
- Clock Range and Accuracy, 11-2

- Clock, Reading the, 11-3
- Clock, setting, 3-22, 11-4
- Clock Time Format, 11-4
- Clock Value, Initial, 11-2
- Clock, Volatile, 11-1
- Closing I/O Path Names, 14-28
- Closure, polygon, 8-54
- CLR I/O (Break) Key, The, 13-15
- CLR I/O** key, 13-15
- CLR I/O key, 9-3
- CMPLX function, 3-20
- Color displays, 8-59
- color graphics, 8-65
- COLOR keyword, 8-68
- color map
 - changing colors, 8-68
 - colors, 8-66
 - default colors, 8-67
 - turning on, 8-66
- COLOR MAP keyword, 8-66
- color models, 8-68
- colors
 - color mapped, 8-66
 - non-color mapped, 8-65
- COM blocks, 6-11-14, 6-23
- COM Blocks, Hints for Using, 6-13
- Comma separator, 15-4
- Comments, 2-8
- Communication, Program/Subprogram, 6-6
- Comparisons Between Two REAL or COMPLEX Values, 3-12
- Comparisons, REAL and COMPLEX Numbers and, 12-4
- Comparisons, Rounding Errors Resulting from, 3-17
- Complement lines, 8-39
- COMPLEX Arguments and the Trigonometric Mode, 3-20
- COMPLEX data type, 3-1, 4-1
- COMPLEX Data Type, 3-2
- Complex Functions, 3-19
- COMPLEX Numbers and Comparisons, REAL and, 12-4
- COMPLEX Numbers, Determining the Parts of, 3-20
- COMPLEX Numbers, Evaluating, 3-19
- COMPLEX statement, 4-2, 4-6
- COMPLEX Values, Creating, 3-19
- COMPLEX variables, 3-4
- Computer backplane, 14-3
- COM statement, 2-30, 2-32, 4-2, 4-6, 7-13
- COM vs. Pass Parameters, 6-12
- Concatenation, String, 5-4
- Concurrency, 20-27
- Conditional execution, 2-9
- Conditional segment, 2-10
- Conditional Segments, Multiple-Line, 2-11
- Conditions, Interrupt, 18-20
- CONJG function, 3-21
- Context Switching, 6-16
- CONTINUE** key, 2-4
- CONTINUE key, 2-6
- Continuing a Program, Pausing and, 13-5
- Control Characters, 10-4
- Control Characters, Displaying, 5-25
- CONTROL KBD statement, 8-64
- CONTROL statement, 17-3, 21-30, 21-32
- CONT statement, 2-4
- Conversion, Case, 5-16
- Conversion, Number-Base, 5-23
- Conversions, Implicit Type, 3-4
- CONVERT ... BY INDEX statement, 19-11
- CONVERT ... BY PAIRS statement, 19-13
- Converting from Rectangular to Polar Coordinates, 3-21
- CONVERT IN statement, 19-14
- CONVERT OUT statement, 19-14
- CONVERT statement, 19-11

- Coordinate system, symbol, 8-22, 8-57
 - Copying an Array into a Subarray, 4-16
 - Copying a Portion of an Array into Itself, 4-19
 - Copying a Subarray into an Array, 4-17
 - Copying a Subarray into another Subarray, 4-18
 - Copying Data into the Destinations, 14-17
 - Copying Data to the Destination, 14-15
 - Copying Entire Arrays into Other Arrays, 4-9
 - Copying Subarrays, 4-13
 - Copying Subarrays, Rules for, 4-20
 - COS function, 3-15
 - COSH function, 3-15
 - COUNT parameter, 20-16
 - Creating COMPLEX Values, 3-19
 - Crossbars, 8-13
 - Cross Reference, Example Program and, 13-6
 - Cross-Reference Listing, Generating a, 13-6
 - Cross References, 13-6
 - CRT Characters, Alternate, 5-27
 - CRT Character Set, 5-13
 - CRT function, 3-24
 - CS (character set) command, 9-9
 - Csize program, 8-20
 - CSIZE statement, 8-19
 - Current relative origin, 8-49
 - Cursor-control routine, 2-26
 - Cycles and Delays, 11-15
- D**
- DATA and READ Statements, Using, 7-2
 - data bits
 - serial I/O, 21-27
 - Datacomm interface (with TRANSFER), 20-41
 - Data Compatibility, 14-5
 - Data-driven plotting, 8-45
 - Data, Entering, 16-1
 - Data Flow, Directing, 14-19
 - Data Handshake, 14-14
 - Data in Programs, Storing, 7-1
 - Data Input by the User, 7-2
 - Data in Variables, Storing, 7-2
 - Data, Outputting, 15-1
 - Data Pointer, Moving the, 7-5
 - Data, Re-Directing, 14-30
 - Data Representations, 14-6
 - DATA statement, 4-8, 6-16, 7-1, 7-3
 - Data Storage and Retrieval, 7-1
 - Data Type, COMPLEX, 3-2
 - Data Type, INTEGER, 3-2
 - Data Type, REAL, 3-1
 - Date and Time of Day, Determining the, 11-3
 - Date format, European, 11-10
 - DATE function, 3-22
 - Date Functions, Time and, 3-22
 - Dates, Days Between Two, 11-13
 - Date, Setting Only the, 11-8
 - DATE\$ string function, 11-3
 - Day of the Week, 11-13
 - Days Between Two Dates, 11-13
 - Day, Time of, 11-16
 - DCOMM binary, 11-2
 - Deactivated interrupt, 2-27
 - Deactivating events, 2-27
 - Debugging Programs, 13-1
 - Declaring Variables, 3-3
 - Default dimensioned length of a string, 5-2
 - Default range, 4-15
 - DEF FN statement, 6-6, 6-25
 - Degrees, 3-15
 - DEG statement, 3-15, 3-20, 6-17, 8-24
 - Delayed Assignment Surprise, 3-11
 - Delays, Cycles and, 11-15
 - DELAY statement, 19-16

- Deleting Subprograms, 6-23, 6-26
 - Delimiter Characters, 20-16
 - DELIM parameter, 20-16
 - DEL LN statement, 6-26
 - DELSUB statement, 6-23
 - Design, Top-Down, 6-28
 - Determining Error Number and Location, 12-7
 - DET function, 3-14
 - Device selector, 10-3
 - device selectors
 - for HP-IB, 21-2
 - Device Selectors, 14-21
 - Device selectors, using, 10-3
 - Digit specifier, 15-15
 - Dimensioning, Problems with Implicit, 4-6
 - DIM statement, 3-3, 4-2
 - Directing Data Flow, 14-19
 - Direct Interface Access, 17-12
 - Direction, label, 8-24
 - Direct memory access, 20-13, 20-37
 - Directories, Extended Access of, 7-18
 - Disable alpha scrolling, 8-64
 - Disabled interrupt, 2-27
 - DISABLE EXT SIGNAL statement, 18-23
 - DISABLE statement, 2-29, 6-18-19, 12-6
 - Disabling Error Trapping (OFF ERROR), 12-6
 - Disabling Events, 2-29
 - display
 - color mapped, 8-66
 - non-color mapped, 8-65
 - Display-enable mask, 8-60
 - Display Enhancement Characters, 5-26
 - dithered colors, 8-65, 8-69
 - Dithering, 8-48
 - DMA Mode, 20-34
 - DMA transfers
 - BASIC/UX, 20-13
 - DOT function, 3-14
 - Double-Subscript Substrings, 5-6
 - Drawing lines, 8-3
 - Drawing Modes, 8-39
 - DROUND function, 3-17-18, 12-4
 - DUMP DEVICE IS statement, 9-2, 10-13
 - Dump graphics, 9-2, 9-14
 - DUMP GRAPHICS key, 9-3
 - DUMP GRAPHICS statement, 9-2
 - DVAL function, 3-23, 5-23
 - DVAL\$ string function, 5-23
 - Dyadic operator, 3-11
 - Dynamically Allocated, Two-Dimensional INTEGER Array, 4-5
 - Dynamically allocate memory, 3-4
- ## E
- EDGE keyword, 8-54, 8-56
 - EDGE parameter, 8-52
 - Edges, screen, 8-5
 - Editing Subprograms, 6-25
 - Electrical and Mechanical Compatibility, 14-4
 - Empty pointer, 20-9
 - Enable alpha scrolling, 8-64
 - ENABLE EXT SIGNAL statement, 18-23
 - ENABLE INTR statement, 21-12, 21-17
 - ENABLE statement, 2-29
 - END in Freefield OUTPUT, 15-9
 - End-of-line (EOL), 15-4
 - End-of-line (EOL) sequences, 7-14
 - End-of-line sequence, 15-7, 19-1, 19-15
 - End-or-identify, 16-12, 16-23
 - End-Or-Identify (EOI) signal, 21-25
 - END parameter, 20-16
 - END SELECT statement, 2-14
 - END statement, 2-4, 6-4
 - END with Data Communications Interfaces, 15-35
 - END with HP-IB Interfaces, 15-10, 15-33

- END with OUTPUTs that Use Images, 15-32
- END with the Data Communications Interface, 15-11
- ENTER and Buffers, OUTPUT and, 20-13
- ENTER images, 15-27
- Entering Data, 16-1
- Entering String Data, 16-8
- ENTER statement, 7-11, 14-13, 14-16-17, 14-20, 16-1, 16-14, 21-3, 21-30, 21-34, 21-40
- Enters that Use Images, 16-14
- ENTER USING statement, 16-14
- EOF pointer, 7-13
- EOI Re-Definition, 16-23
- Equal units (isotropic scaling), 8-5, 8-16
- ERRDS function, 12-8
- ERRL function, 12-7, 12-15
- ERRL in Subprograms, Using ERRLN and, 12-12
- ERRLN and ERRL in Subprograms, Using, 12-12
- ERRLN function, 12-7, 12-14
- ERRM\$ string function, 12-8, 12-14
- ERRN function, 12-7, 12-14
- Error Conditions, Clearing, 12-16
- Error detection, 9-9
- Error, Example of Simulating an, 12-15
- Error Number and Location, Determining, 12-7
- Error Reporting, 20-29
- Error Responses, Overview of, 12-1
- ERROR RETURN statement, 12-8
- errors
 - serial I/O, 21-29, 21-35
- Errors, Anticipating Operator, 12-2
- Errors, Handling, 12-1
- Errors with BASIC Programs, Trapping, 12-5
- Error Trapping and Recovery, Scope of, 12-6
- Error Trapping (OFF ERROR), Disabling, 12-6
- Escape-Code Sequences, 10-5
- European date format, 11-10
- Evaluating COMPLEX Numbers, 3-19
- Evaluating Scalar Expressions, 3-9
- Evaluating String Expressions, 5-3
- Evaluation Hierarchy, 5-3
- Event-checking, 2-21
- Event-initiated branching, 2-2, 2-21, 2-24
- Event-Initiated Branching, 18-1
- Event-initiated RECOVER statement, 6-18
- Events, Branching on Clock, 11-14
- Events, Disabling, 2-29
- Events, Enabling Interrupt, 18-16
- Events, Logging and Servicing, 18-6
- Events, Servicing Pending, 18-13
- Events, Types of, 2-22, 18-1
- Example programs, 8-16
 - CharCell, 8-21
 - Csize, 8-20
 - Gstore, 8-44
 - Iplot, 8-53
 - Ldir, 8-25
 - Lem1, 8-45
 - Lorg, 8-23
 - Pen, 8-39
 - Rplot, 8-50
 - SinGrdAxes, 8-35
 - SinLabel, 8-27
 - SinLabel2, 8-28
 - SinViewprt, 8-17
 - Symbol, 8-57
- Example programs disk, 8-1
- Example statements
 - LINE TYPE, 8-42
- Executing Commands While a Program Is Running, 13-2

- Execution Speed, 14-29
- EXIT IF statement, 2-20
- Expanded graphics dumps, 9-4
- EXP function, 3-14
- Explicitly close, 14-28
- Exponential Functions, 3-14
- Exponent specifier, 15-15
- Expressions as Pass Parameters, 3-12
- Expressions, hierarchy for, 3-9
- Extended Access of Directories, 7-18
- extended addressing, 21-7
- Extended Character Set, The, 5-26
- External Printer, Using the, 10-4

- F**
- File Access, A Closer Look at General, 7-11
- File Input and Output, 7-6
- Files Cataloged, Getting a Count of, 7-21
- Files, Cataloging Selected, 7-22
- Files, Getting a Count of Selected, 7-24
- File specifiers, 7-10
- Files, plotting to, 9-10
- Files, Skipping Selected, 7-24
- Files to the Spooler Directories, Writing, 10-13
- File Types, Brief Comparison of Available, 7-7
- fill colors, 8-65, 8-69
- FILL keyword, 8-54, 8-56
- FILL parameter, 8-52
- Fill pointer, 20-9
- FIND statement, 13-6
- Firmware, 14-13
- Floor of a number, 8-6
- FNEND statement, 6-24, 6-26
- FN statement, 2-6
- Force, pen, 9-9
- FOR ... NEXT structure, 2-15, 2-17
- Formal parameter lists, 6-6, 6-8
- FORMAT attribute, 7-14
- FORMAT attributes, 7-14, 19-1
- FORMAT Attributes, Assigning Default, 19-4
- Format, Clock Time, 11-4
- FORMAT OFF statement, 7-13-14, 14-31, 19-2
- FORMAT ON statement, 7-13, 14-31, 19-2
- FORMAT statement, 19-1
- Formatted Printing, 10-5
- Formatting Arrays for Display, 4-11
- Formatting, Transfer, 20-14
- Four-dimensional array, 4-5
- FRACT function, 3-13
- Free-Field Enters, 16-1
- Free-Field ENTER Statements, 16-11
- Free-field output, 15-1
- Freefield OUTPUT, END in, 15-9
- FS (force select) command, 9-9
- Full-Mode handshake, 21-39
- Function, ABS, 3-13
- Function, ACS, 3-15
- Function, ACSH, 3-15
- Function and a Subprogram, Difference, 6-5
- Function, ASN, 3-15
- Function, ASNH, 3-15
- Function, ATN, 3-15
- Function, ATNH, 3-15
- Function, BASE, 3-14
- Function, BINAND, 3-16
- Function, BINCOMP, 3-16
- Function, BINEOR, 3-16
- Function, BINIOR, 3-16
- Function, BIT, 3-16
- Function, CMPLX, 3-20
- Function, CONJG, 3-21
- Function, COS, 3-15
- Function, COSH, 3-15
- Function, CRT, 3-24
- Function, DATE, 3-22

- Function, DET, 3-14
 - Function, DOT, 3-14
 - Function, DROUND, 3-17-18, 12-4
 - Function, DVAL, 3-23, 5-23
 - Function, ERRDS, 12-8
 - Function, ERRL, 12-7, 12-15
 - Function, ERRLN, 12-7, 12-14
 - Function, ERRN, 12-7, 12-14
 - Function, EXP, 3-14
 - Function, FRACT, 3-13
 - Function, IMAG, 3-21
 - Function, INT, 3-13
 - Function, IVAL, 3-23, 5-23
 - Function, KBD, 3-24
 - Function, LGT, 3-14
 - Function, LOG, 3-14
 - Function, MAX, 3-16
 - Function, MAXREAL, 3-13
 - Function, MIN, 3-16
 - Function, MINREAL, 3-13
 - Function, NUM, 5-12
 - Function, PI, 3-15
 - Function, POS, 5-10, 5-12
 - Function, PROUND, 3-17
 - Function, PRT, 3-24
 - Function, RANK, 3-14
 - Function, REAL, 3-21
 - Function, RES, 3-24
 - Function, RND, 3-18, 8-4, 8-6
 - Function, ROTATE, 3-16
 - Functions
 - RATIO, 8-16
 - RND, 8-4, 8-6
 - Functions, Arithmetic, 3-13
 - Functions, Array, 3-14
 - Functions, Base Conversion, 3-23
 - Functions, Binary, 3-16
 - Function, SC, 3-24
 - Functions, Complex, 3-19
 - Functions, Exponential, 3-14
 - Functions, General, 3-24
 - Function, SGN, 3-13
 - Function, SHIFT, 3-16
 - Functions, Hyperbolic, 3-15
 - Function, SIN, 3-15
 - Function, SINH, 3-15
 - Function, SIZE, 3-14
 - Functions, Numerical, 3-13
 - Function, SQR, 3-13
 - Function, SQRT, 3-13, 3-19
 - Functions, Rounding, 3-16
 - Functions, String, 5-14
 - Functions, String-Related, 5-9
 - Functions, Subprograms and User-Defined,
 - 6-1
 - Functions, Time and Date, 3-22
 - Functions, Trigonometric, 3-14
 - Function, SUM, 3-14
 - Function, TAN, 3-15
 - Function, TANH, 3-15
 - Function, TIME, 3-22, 11-6
 - Function, TIMEDATE, 3-22, 11-3, 11-6
 - Function, VAL, 5-11
- G**
- GDUs, 8-9, 8-16
 - General File Access, A Closer Look at, 7-11
 - General Functions, 3-24
 - Generating a Cross-Reference Listing, 13-6
 - GESCAPE statement, 8-43, 8-59
 - GET statement, 2-30-32, 6-14
 - GET, Using, 2-30
 - GINIT statement, 8-4
 - GLOAD command, 8-44
 - GOSUB statement, 2-6-7, 6-16, 6-18
 - GOTO statement, 2-6-7, 6-16, 6-18
 - GPIO interface
 - checking peripheral, 21-39, 21-43
 - control lines, 21-36
 - data logic sense, 21-37

- data transfers, 21-40
 - enabling interrupts, 21-44
 - flexible interface, 21-36
 - handshake logic sense, 21-38
 - handshake methods, 21-38
 - handshake modes, 21-39
 - input lines, 21-36
 - interface ready register, 21-46
 - interrupt enable register, 21-45
 - interrupt events, 21-44
 - interrupt priority, 21-37
 - output lines, 21-36
 - peripheral status register, 21-46
 - processing interrupts, 21-46
 - resetting, 21-40
 - select codes, 21-37
 - timeouts, 21-43
 - GPIO interface (with TRANSFER), 20-40
 - GRAPH BIN file, 8-1
 - graphics
 - changing color map, 8-68
 - color mapped, 8-66
 - color models, 8-68
 - colors, 8-65
 - default color map, 8-67
 - fill colors, 8-65, 8-69
 - label colors, 8-65
 - non-color mapped, 8-65
 - Graphics
 - Binary files, 8-1
 - Character size, 8-19
 - Display-Enable mask, 8-60
 - Dumps, 9-2, 9-14
 - Example programs, 8-1
 - GRID vs. AXES, 8-33
 - Initialization, 8-4
 - Labels, 8-11, 8-19
 - Line types, 8-42
 - Plotting, 9-7
 - Rotation, 8-49
 - Scaling, isotropic, 8-5, 8-16
 - Statement requirements, 8-1
 - Storing and retrieving images, 8-43
 - Translation, 8-49
 - Turning on, 8-4
 - Write-Enable mask, 8-59
 - Graphics/Alpha interaction, 8-61
 - Graphics and alpha, separate, 8-4
 - Graphics Display Units, 8-9, 8-16
 - GRAPHICS ON statement, 8-4
 - GRAPHX BIN file, 8-1
 - GSEND statement, 9-8, 9-11
 - Gstore program, 8-44
 - GSTORE statement, 8-43
- ## H
- Halting Program Execution, 2-3
 - Handling Errors, 12-1
 - handshake
 - serial I/O, 21-24, 21-35
 - Handshake, Data, 14-14
 - Hard clip limits, 8-9, 8-38
 - Hardware priority, 18-11
 - Hewlett-Packard Graphics Language, 9-8
 - HFS "Extended" Catalog, Getting an, 7-20
 - Hierarchical File System (HFS), 7-8
 - Hierarchy, Evaluation, 5-3
 - Hierarchy for expressions, 3-9
 - Hierarchy, Math, 3-10
 - HPGL commands
 - CS (character set), 9-9
 - Error detection, 9-9
 - FS (force select), 9-9
 - Fundamentals, 9-8
 - Termination, 9-8
 - VS (velocity select), 9-8
 - HP-IB control lines, 21-23
 - HP-IB devices
 - clearing, 21-10
 - configuring parallel poll, 21-13

- initializing, 21-10
 - local lockout, 21-9
 - Local mode, 21-8
 - parallel poll, 21-13
 - Remote mode, 21-8
 - requesting service, 21-11
 - serial poll, 21-14
 - service requests, 21-14
 - status bytes, 21-14
 - triggering, 21-10
 - unconfiguring parallel poll, 21-13
 - HP-IB Device Selectors, 14-24
 - HP-IB interface
 - aborting operations, 21-11
 - address, 21-3, 21-14
 - bus control, 21-7
 - changing address, 21-15
 - clearing devices, 21-10
 - configuring parallel poll, 21-13
 - controller status, 21-14
 - controller status and address register, 21-15
 - control lines, 21-23
 - data transfers, 21-3
 - device selectors, 21-2
 - enabling service requests, 21-12, 21-20
 - extended addressing, 21-7
 - interrupt enable mask register, 21-18
 - interrupt-enable register, 21-12
 - interrupt levels, 21-2
 - multiple listeners, 21-6
 - multiple service requests, 21-12
 - not active controller, 21-14, 21-16
 - operation, 21-3
 - parallel poll, 21-13
 - passing control, 21-16
 - programming information, 21-1
 - requesting service, 21-21
 - response to parallel poll, 21-21
 - response to serial poll, 21-22
 - secondary addresses, 21-7
 - select codes, 21-3
 - serial poll, 21-14
 - service request processing, 21-12
 - setting local lockout, 21-9
 - triggering devices, 21-10
 - unconfiguring parallel poll, 21-13
 - HP-IB interface (with TRANSFER), 20-41
 - HP-IB references, 21-26
 - HP Raster Interface Standard, 9-2
 - HP-UX file, 7-14
 - HP-UX printer spooler, using, 10-15
 - HSL color model, 8-68
 - hue, 8-68
 - Hyperbolic Functions, 3-15
- I**
- IDRAW statement, 8-53
 - IEEE 488 standard, 21-4
 - IF ... THEN ... ELSE statement, 2-13
 - IF ... THEN statement, 2-10
 - IF ... THEN structure, 2-20
 - Image Definitions During Outputs, 15-15
 - Image output, 15-2
 - Image OUTPUT, 15-1
 - Image Repeat Factors, 15-29
 - Image Re-Use, 15-30, 16-27
 - Images, 15-13, 16-15
 - Images, binary, 15-23
 - Images, ENTER, 15-27
 - Images, negative, 9-6
 - Images, nested, 15-31
 - Images, numeric, 15-15
 - Images, Outputs that Use, 15-12
 - Image Specifiers, Additional, 10-11
 - Image Specifiers, Numeric, 10-8
 - Image Specifiers, String, 10-10
 - Images, Special-Character, 15-25
 - Images, storing and retrieving, 8-43
 - Images, string, 15-21

- Images, Terminating Enters that Use, 16-22
- Images, Using, 10-7
- IMAG function, 3-21
- IMOVE statement, 8-53
- Implicit Dimensioning, Problems with, 4-6
- Implicit Type Conversions, 3-4
- Inbound and Outbound Transfers, 20-2
- Inbound transfer, 20-2
- Incremental plotting, 8-53
- Individual Array Elements, Using, 4-7
- Infinite loop, 2-7, 2-24
- Initial Clock Value, 11-2
- Initialization, graphics, 8-4
- Initialization, Variable, 6-17
- Input, 14-2
- INPUT statement, 5-15, 7-2, 13-10
- Inserting Subprograms, 6-25
- Installing a printer, 10-1
- INTEGER data type, 3-1-2, 4-1
- Integers, ASCII Representation of, 14-10
- Integers, Internal Representation of, 14-9
- Integers, Representing Signed, 14-8
- INTEGER statement, 4-2, 4-6
- INTENSITY keyword, 8-68
- Interface Access, Direct, 17-12
- Interface Functions, Additional, 14-5
- Interface Interrupts, 18-15
- Interface, primary function of an, 14-3
- interface registers
 - reference information, 21-1
- Interface Registers, 17-2
- Interface select code, 10-3
- Interfaces, select codes, 14-22
- Interfaces, Select Codes of Optional, 14-23
- Interface Timeouts, 18-21
- Interfacing Concepts, 14-1
- Internal Numeric Formats, 3-8
- Internal real-time clock, 11-15
- Internal Representation of Integers, 14-9
- Internal Representation of Real Numbers, 14-11
- Interrupt Conditions, 18-20
- Interrupt, deactivated, 2-27
- Interrupt, disabled, 2-27
- interrupts
 - GPIO, 21-44
- Interrupts and Timeouts, 18-1
- Interval Timing, 11-13
- INT function, 3-13
- INT Mode, 20-35
- Inverse video images, 9-6
- I/O, 14-2
- IO binary, 21-1
- I/O Examples, 14-14
- I/O path, 7-11
- I/O Path Attributes, 19-1
- I/O Path Attributes, Specifying, 19-4
- I/O Path Benefits, 14-29
- I/O path name, 14-25, 14-27, 17-8, 19-1, 20-8
- I/O path names, 7-10, 7-12
- I/O Path Names Assigned to a BDAT File, 17-9
- I/O Path Names Assigned to a Buffer, 17-11
- I/O Path Names Assigned to a Device, 17-8
- I/O Path Names Assigned to a DOS File, 17-10
- I/O Path Names Assigned to an ASCII File, 17-8
- I/O Path Names Assigned to an HP-UX File, 17-9
- I/O Path Names, Closing, 14-28
- I/O Path Names in Subprograms, 14-29
- I/O Path Names, Re-Assigning, 14-28
- I/O Path Names to Named Buffers, Assigning, 20-7

- I/O Path Names to Unnamed Buffers, Assigning, 20-7
- I/O Path, Opening an, 7-12
- I/O Path Registers, 17-4
- I/O Path Register Summary, 17-8
- I/O Paths, Closing, 7-15
- I/O Process, 14-12
- I/O Statements and Parameters, 14-12
- Iplot program, 8-53
- IPLOT statement, 8-48, 8-53
- Isotropic scaling, 8-5, 8-16
- Item Separators, 15-3, 16-2
- Item Terminators, 15-3, 16-2
- IVAL function, 3-23, 5-23
- IVAL\$ string function, 5-23

- K**
- KBD function, 3-24
- Keyboard, Calling Subprograms from the, 6-20
- Keyboard Commands Disallowed During Program Execution, 13-5
- Keywords
 - AREA COLOR, 8-54, 8-56
 - AREA INTENSITY, 8-48, 8-54, 8-56
 - AREA PEN, 8-54, 8-56
 - BINCMP, 9-6
 - CLIP, 8-38
 - CLIP OFF, 8-9, 8-31
 - CONTROL KBD, 8-64
 - CSIZE, 8-19
 - DEG, 8-24
 - DUMP DEVICE IS, 9-2
 - DUMP GRAPHICS, 9-2
 - EDGE, 8-54, 8-56
 - EXPANDED (DUMP DEVICE IS), 9-4
 - FILL, 8-54, 8-56
 - GESCAPE, 8-43, 8-59
 - GINIT, 8-4
 - GLOAD, 8-44
 - GRAPHICS ON, 8-4
 - GSEND, 9-8, 9-11
 - GSTORE, 8-43
 - IDRAW, 8-53
 - IMOVE, 8-53
 - IPLOT, 8-48, 8-53
 - LABEL, 8-11, 8-19, 8-26
 - LDIR, 8-24
 - LORG, 8-23
 - MERGE ALPHA WITH GRAPHICS, 8-62
 - MOVE, 8-11
 - OUTPUT, 9-3
 - PDIR, 8-49, 8-54
 - PIVOT, 8-49, 8-54
 - PLOT, 8-3, 8-11, 8-45, 8-48
 - PLOTTER IS, 9-8
 - POLYGON, 8-54
 - POLYLINE, 8-55
 - RAD, 8-24
 - RATIO, 8-16
 - RECTANGLE, 8-55
 - RND, 8-4, 8-6
 - RPLOT, 8-49
 - SEPARATE ALPHA FROM GRAPHICS, 8-62
 - SHOW, 8-5, 8-16
 - SYMBOL, 8-48, 8-56
 - VIEWPORT, 8-9-10, 8-16, 8-38
 - WINDOW, 8-16, 8-21
- Keywords AXES, 8-13
- Knob, Example of Using, 2-26

- L**
- Label direction, 8-24
- LABEL, keyword, 2-25
- Label origin, 8-23
- Labels, 8-11, 8-19
- Labels, bold, 8-28
- LABEL statement, 8-11, 8-19, 8-26
- Ldir program, 8-25

- LDIR statement, 8-24
- Lem1 program, 8-45
- LET statement, 3-4, 7-2
- Letters, bold, 8-28
- LEX binary, 5-5
- Lexical Order, Introduction to, 5-24
- LEXICAL ORDER IS statement, 5-5, 5-16, 5-24-25, 5-28
- Lexical Order, Predefined, 5-28
- LGT function, 3-14
- Libraries, Using Subprograms, 6-20
- LIF file, 7-13
- Limits
 - Hard clip, 8-9, 8-38
 - screen, 8-5
 - Screen, 8-9
 - Soft clip, 8-9, 8-38
- Linear flow, 2-2
- Line labels, 2-8
- Lines, drawing, 8-3
- Line types, 8-41
- LINE TYPE statement, 8-42
- LINPUT statement, 5-15, 7-2, 13-10
- listeners
 - multiple, 21-6
- LIST statement, 21-34
- Live Keyboard, Using, 13-2
- LOAD command, 2-30
- Loading Several Subprograms at Once, 6-22
- Loading Subprograms, 6-21
- Loading Subprograms One at a Time, 6-21
- Loading Subprograms Prior to Execution, 6-22
- LOAD statement, 2-30, 6-14, 6-23
- LOADSUB ... FROM statement, 6-21, 6-23
- LOADSUB statement, 6-22, 11-7
- Local Lockout mode
 - setting, 21-9
- LOCAL LOCKOUT statement, 21-7, 21-9
- Local mode
 - setting, 21-8
- LOCAL statement, 21-7, 21-9
- Locking an interface during a TRANSFER, 20-36
- LOG function, 3-14
- Loop counter, 2-15
- LOOP ... END LOOP structure, 2-19
- Loop iterations, conditional, 2-17
- Loop iterations, fixed, 2-17
- Loop iterations formula, 2-15
- LOOP statement, 2-20
- Lorg program, 8-23
- LORG statement, 8-23
- luminosity, 8-68
- LWC\$ string function, 5-16, 5-24

M

- Machine Limits, 3-7
- Magnitude, 3-22
- Manual Examples disk, 8-1, 8-16
- Manual Organization, 1-1
- Marks, tick, 8-13
- Mask
 - Alpha, 8-60
 - Display-enable, 8-60
 - Write-enable, 8-59
- masks
 - interrupt enable, 21-12, 21-18
- MASS STORAGE IS statement, 7-12
- MAT binary, 4-1
- MAT BIN file, 8-16
- Math Hierarchy, 3-10
- MAT SEARCH statement, 5-23
- MAT SORT statement, 5-17, 5-24
- MAT statement, 4-6, 4-9, 5-16
- MAX function, 3-6, 3-16
- MAXREAL function, 3-13

- Mechanical Compatibility, Electrical and, 14-4
- MERGE ALPHA WITH GRAPHICS statement, 8-62
- Merging Subprograms, 6-26
- MIN function, 3-6, 3-16
- MINREAL function, 3-13
- Modes, drawing, 8-39
- Modifiers, Statement-Termination, 16-24
- Monadic operator, 3-11
- Monochromatic pens, 8-39
- MOVELINES statement, 6-25
- MOVE statement, 8-11
- Moving the Data Pointer, 7-5
- Multi-plane bit-mapped displays, 8-59
- Multiple-Field Numeric Image Specifiers, 10-9
- Multiple-Line Conditional Segments, 2-11
- Multiple picture display, 8-63
- Multiple Termination Conditions, 20-17
- Multiplexer device selectors, 14-24
- Multi-tasking environment, 14-18
- Multi-user environment, 14-18
- N**
- Named buffer, 20-6
- Named Buffers, Assigning I/O Path Names to, 20-7
- Named Buffers, Creating, 20-6
- Named Buffers via Variable Names, Accessing, 20-11
- Names, of variables (syntax of), 14-25
- Names, string-variable, 14-20
- Naming variables, 3-3
- Negative images, 9-6
- Nested constructs, 2-12
- Nested Images, 15-31, 16-27
- NO HEADER statement, 7-21
- non-active controller, 21-14, 21-16
- Non-Repeatable Specifiers, 16-26
- Non-square pixels, 9-4
- Non-volatile clock, 11-1
- Normal program flow, 2-22
- Null string, 5-1
- Number
 - Ceiling, calculating, 8-6
 - Floor, calculating, 8-6
- Number-Base Conversion, 5-23
- Number builder, 16-3
- Numbers, Representing, 14-7
- Numerical Functions, 3-13
- Numeric Arrays, 4-1
- Numeric Computation, 3-1
- Numeric Data Types, 3-1
- Numeric Formats, Internal, 3-8
- Numeric Format, Standard, 15-2
- Numeric Images, 15-15, 16-17
- Numeric Image Specifiers, 10-8
- Numeric Image Specifiers, Examples of, 10-9
- Numeric Image Specifiers, Multiple-Field, 10-9
- Numeric specifier, 16-17
- Numeric-to-String Conversion, 5-12
- Numeric variables, 3-2
- NUM function, 5-12
- O**
- OFF CYCLE statement, 11-15
- OFF DELAY statement, 11-15
- OFF-event, 2-27
- OFF EXT SIGNAL statement, 18-23
- OFF KEY statement, 2-27
- OFF KNOB statement, 2-27
- OFF TIME statement, 11-15
- ON CDIAL statement, 2-23, 18-1
- ON CYCLE statement, 2-23, 11-14
- ON DELAY statement, 2-23, 11-14
- One-dimensional array, 4-1
- ON ... CALL statement, 6-24
- ON ... event statement, 2-22
- ON ... RECOVER statement, 6-24

- ON END statement, 2-23, 18-2
 - ON EOR statement, 2-23
 - ON EOT statement, 2-23
 - ON ERROR branching, 12-6
 - ON ERROR CALL, A Closer Look At, 12-11
 - ON ERROR Execution at Run-Time, 12-6
 - ON ERROR GOSUB, A Closer Look at, 12-8
 - ON ERROR GOTO, A Closer Look At, 12-9
 - ON ERROR Priority, 12-6
 - ON ERROR RECOVER, A Closer Look At, 12-13
 - ON ERROR statement, 2-23, 18-2
 - ON-event, 2-27
 - ON-event statement, 2-21
 - ON EXT SIGNAL statement, 2-23, 18-2, 18-23
 - ON HIL EXT statement, 2-23, 18-2
 - ON INTR statement, 2-23, 18-2
 - ON KBD statement, 2-23
 - ON KEY statement, 2-23-25, 6-17-18, 18-2
 - ON KNOB statement, 2-23, 2-27, 18-2
 - ON SIGNAL statement, 2-23
 - ON TIMEOUT statement, 2-23, 10-11, 18-2, 18-23, 21-43
 - ON TIME statement, 2-23, 11-14
 - Operand array, 4-9
 - Operator, dyadic, 3-11
 - Operator Errors, Anticipating, 12-2
 - Operator, monadic, 3-11
 - Operator, relational, 3-11
 - Operators, 3-11
 - OPTIONAL/NPAR combination, 6-10
 - OPTIONAL parameter, 6-9, 6-11
 - OPTION BASE statement, 3-3, 4-2, 5-17, 6-17
 - Origin, label, 8-23
 - Origin, relative, 8-49
 - Outbound transfer, 20-2
 - Outbound Transfers, Inbound and, 20-2
 - Output, 14-2
 - OUTPUT and ENTER and Buffers, 20-13
 - Output, Appearance of, 10-14
 - OUTPUT statement, 2-10, 9-3, 14-13-14, 14-20, 15-1, 16-1, 21-3, 21-30, 21-34, 21-40
 - Outputs that Use Images, 15-12
 - Outputting Data, 15-1
 - OUTPUT USING statement, 15-12
- P**
- PAIRS conversions, 19-13
 - Paper size (plotter), 9-10
 - parallel poll, 21-12, 21-21
 - Parameter Lists, Formal, 6-6
 - Parameters, Expressions as Pass, 3-12
 - Parameters Lists, 6-6
 - Parameters, OPTIONAL, 6-9
 - Parameters passed by reference, 3-6
 - Parameters passed by value, 3-6
 - parity
 - serial I/O, 21-33
 - parity bit
 - serial I/O, 21-27
 - Parity Generation and Checking, 19-16
 - PARITY statement, 19-17
 - PASS CONTROL statement, 21-16
 - Passing by Value vs. Passing By Reference, 6-7
 - passing control, 21-16
 - Passing Entire Arrays, 4-13
 - Pass parameter lists, 6-7, 6-9
 - Pass Parameters, COM vs., 6-12
 - Pass Parameters, Expressions as, 3-12
 - Path name, I/O, 14-27
 - PAUSE statement, 2-5
 - Pausing and Continuing a Program, 13-5

- PDEV binary, 6-25
- PDIR statement, 8-49, 8-54
- pen
 - colors, 8-65
- Pen-control parameters, 8-45, 8-57
- Pen force, controlling, 9-9
- pen numbers
 - default color map colors, 8-67
 - non-color mapped colors, 8-66
 - setting, 8-65
- Pen program, 8-39
- Pens, 8-39
- Pen speed, controlling, 9-8
- PEN statement, 8-65
- PI function, 3-15
- PIVOT statement, 8-49, 8-54
- Pixels, 8-39, 8-48
- Planes of a Three-Dimensional REAL
 - Array, 4-3
- Plot labeling, 8-11, 8-19
- PLOT statement, 8-3, 8-11, 8-45, 8-48
- PLOTTER IS statement, 9-8
- Plotter paper size, 9-10
- Plotters, 9-7
- Plotting
 - Character set selection, 9-9
 - Files, plotting to, 9-10
 - Fundamentals, 9-7
 - HPGL commands, 9-8
 - Limitations, 9-11
 - Paper size specification, 9-10
 - Pen force, 9-9
 - Pen speed, 9-8
 - PLOTTER IS statement, 9-8
 - SRM, 9-12
- Plotting, data driven, 8-45
- Plotting, incremental, 8-53
- Pointer, Moving the Data, 7-5
- Pointers, Buffer, 20-9
- Polar Coordinates, Converting from Rect-
angular to, 3-21
- Polygon rotation, 8-54
- Polygons, 8-54
- POLYGON statement, 8-54
- POLYLINE statement, 8-55
- POS function, 2-19-20, 5-10, 5-12
- PPOLL CONFIGURE statement, 21-13
- PPOLL function, 21-7, 21-13
- PPOLL UNCONFIGURE statement, 21-
13
- Premature termination of TRANSFERs,
20-42
- Primary address, 14-24
- primary addresses
 - setting, 21-2
- Primary function of an interface, 14-3
- PRINTALL IS statement, 10-13, 13-14
- Printer Control Characters, 10-4
- Printer, installing, 10-1
- PRINTER IS device, 4-11
- PRINTER IS statement, 10-1
- Printers, non-standard, 9-5
- Printer, system, 10-1
- Printer, Using a, 10-1
- Printer, Using the External, 10-4
- Printing Arrays, 4-11
- Printing graphics, 9-2, 9-14
- PRINT TAB statement, 10-6
- PRINT TABXY statement, 10-6
- Priority, Changing System, 18-8
- Priority, Hardware, 18-11
- Priority, ON ERROR, 12-6
- Priority Restrictions, 11-17
- Priority, Software, 18-7
- Program counter, 2-2, 2-6, 2-8
- Program Design, 6-19
- Program flow, 2-2
- Programs
 - CharCell, 8-21
 - Csize, 8-20
 - Gstore, 8-44
 - Iplot, 8-53

- Ldir, 8-25
 - Lem1, 8-45
 - Lorg, 8-23
 - Pen, 8-39
 - Rplot, 8-50
 - SinGrdAxes, 8-35
 - SinLabel, 8-27
 - SinLabel2, 8-28
 - SinViewprt, 8-17
 - Symbol, 8-57
 - Programs, Debugging, 13-1
 - Programs, example, 8-1, 8-16
 - Program, Single-Stepping a, 13-10
 - Program/Subprogram Communication, 6-6
 - Program-to-Program Communication, 2-32
 - Program Variables, Using, 13-2
 - PROUND function, 3-17
 - PRT function, 3-24
 - Pulse-Mode handshake, 21-39
- R**
- Radians, 3-15
 - Radix specifier, 15-15
 - RAD statement, 3-15, 3-20, 6-17, 8-24
 - RANDOMIZE statement, 3-18
 - Random Number Function, 3-18
 - RANK function, 3-14, 4-6
 - Raster images, dumping, 9-2, 9-14
 - Raster Interface Standard, 9-2
 - Ratio, aspect, 8-16, 8-23
 - RATIO function, 8-16
 - Reading the Clock, 11-3
 - READ statement, 4-8, 7-1, 7-3
 - READ Statement to Fill an Entire Array, Using the, 4-8
 - REAL and COMPLEX Numbers and Comparisons, 12-4
 - REAL data type, 3-1, 4-1
 - REAL Data Type, 3-1
 - REAL function, 3-21
 - Real Numbers, ASCII Representation of, 14-12
 - Real Numbers, Internal Representation of, 14-11
 - Real Numbers, Representing, 14-11
 - REAL statement, 4-2, 4-6
 - Real-Time Clock, 11-1
 - Real-time clock, internal, 11-15
 - Real-time programming, 2-26
 - Re-Assigning I/O Path Names, 14-28
 - RECORDS parameter, 20-17
 - Records, Transferring, 20-17
 - RECOVER statement, 6-16-18, 6-24
 - RECOVER Statement, Subprograms and the, 6-18
 - Recovery, Scope of Error Trapping and, 12-6
 - Rectangles, 8-55
 - RECTANGLE statement, 8-55
 - Rectangular to Polar Coordinates, Converting from, 3-21
 - Recursion, 6-27
 - Redimensioning Arrays, 4-21
 - Redimensioning, Automatic, 4-10
 - REDIM statement, 4-6, 4-21
 - Re-Directing Data, 14-30
 - Reference, Pass by, 6-7
 - References, Cross, 13-6
 - Registers, 14-13, 17-1
 - Interface, 17-2
 - I/O Path, 17-4
 - Registers, Buffer-Type, 20-8
 - Relational Operations, 5-4
 - Relational operator, 3-11
 - Relative origin, 8-49
 - Remote mode
 - setting, 21-8
 - REMOTE statement, 21-8
 - REN statement, 6-25
 - Reordering arrays, 5-21

- Repeatable specifier, 15-29, 16-26
 - REPEAT ... UNTIL structure, 2-15, 2-17-18, 2-20
 - Repeat Factors, 16-26
 - Repeat Factors, Image, 15-29
 - REPEAT statement, 2-18
 - Repeat, String, 5-15
 - Representing Real Numbers, 14-11
 - REQUEST statement, 21-21
 - RESET statement, 20-21
 - RES function, 3-24
 - Resource, specifying a, 14-20
 - RESTORE statement, 7-5
 - RETURN attribute, 19-18
 - RETURN stack, 6-16
 - RETURN statement, 2-7, 12-8
 - Reverse, String, 5-14
 - REV\$ string function, 5-14
 - RGB color model, 8-68
 - rmbxfr**, 20-36
 - RND function, 3-18, 8-4, 8-6
 - ROTATE function, 3-16
 - Rotating a drawing, 8-49
 - Rotation, polygon, 8-54
 - Rounding Errors Resulting from Comparisons, 3-17
 - Rounding Functions, 3-16
 - Rounding problem, 3-6
 - Rplot program, 8-50
 - RPLOT statement, 8-49
 - RPT\$ string function, 5-15
 - Rules for Copying Subarrays, 4-20
 - RUN command, 6-1
 - RUN** key, 2-4
 - Run-Time, ON ERROR Execution at, 12-6
- S**
- saturation, 8-68
 - SAVE statement, 6-23, 7-1, 10-13
 - Scalar Expressions, Evaluating, 3-9
 - Scaling
 - Anisotropic, 8-7
 - Anisotropic, 8-55
 - Isotropic, 8-5, 8-16
 - X-axis, 8-6, 8-16
 - Y-axis, 8-7, 8-16
 - Scaling consideration, 8-18
 - SC function, 3-24
 - Scope of Error Trapping and Recovery, 12-6
 - Screen, defining edges, 8-5
 - Screen dumps, 9-2, 9-14
 - Screen limits, 8-9
 - Scrolling, alpha, 8-64
 - Searching for Strings, 5-22
 - Searching String Arrays, 5-22
 - secondary addresses
 - description, 21-7
 - Sector size, 20-30
 - select codes
 - GPIO interface, 21-37
 - HP-IB interfaces, 21-3
 - Select codes (of built-in interfaces), 14-22
 - Select Codes of Optional Interfaces, 14-23
 - SELECT constructs, 2-13
 - Selection, 2-9
 - Selectors, Device, 14-21
 - Selectors, HP-IB Device, 14-24
 - SELECT statement, 2-13
 - Semicolon separator, 15-5
 - SEND statement, 21-8
 - Separate alpha and graphics, 8-4
 - SEPARATE ALPHA FROM GRAPHICS statement, 8-62
 - Separator, Comma, 15-4
 - Separator, semicolon, 15-5
 - serial interface
 - baud rate, 21-33
 - buffering input, 21-35
 - character length, 21-33
 - data transfers, 21-34

- default parameters, 21-32
- detecting errors, 21-29
- frame format, 21-27, 21-33
- handshaking, 21-35
- parity, 21-28, 21-33
- processing errors, 21-35
- programming information, 21-26
- resetting, 21-32
- setting parameters, 21-32
- stop bits, 21-33
- Serial interface (with TRANSFER), 20-40
- serial poll, 21-22
- service requests
 - device status, 21-14
 - enabling, 21-12, 21-20
 - from HP-IB interface, 21-21
 - multiple, 21-12
 - operation, 21-11
 - processing, 21-12
- Service Requests, 18-19
- service request (SRQ) line, 21-25
- Service routine, 18-6
- Service Routines, Setting Up Error, 12-5
- SET PEN statement, 8-68
- SET TIMEDATE statement, 3-22, 11-6
- SET TIME statement, 3-22
- Setting Only the Date, 11-8
- Setting Only the Time, 11-6
- Setting the clock, 3-22, 11-4
- Setting Up Error Service Routines, 12-5
- SGN function, 3-13
- Shared Printer, Sending Program Output to a, 10-14
- SHIFT function, 3-16
- Shift-in control character, 2-19
- Shift-out control character, 2-19
- SHOW statement, 8-5, 8-16
- Signed Integers, Representing, 14-8
- Sign specifier, 15-15
- Simple Branching, 2-6
- Simulating an Error, Example of, 12-15
- Simulating Errors (CAUSE ERROR), 12-14
- Simultaneous TRANSFERs, 20-40
- SIN function, 3-15
- Single-Stepping a Program, 13-10
- Single-Subscript Substrings, 5-5
- SinGrdAxes program, 8-35
- SINH function, 3-15
- SinLabel2 program, 8-28
- SinLabel program, 8-27
- SinViewprt program, 8-17
- SIZE function, 3-14, 4-6
- Soft clip limits, 8-9, 8-38
- Softkeys, Subprograms and, 6-17
- Software priority, 18-4, 18-7
- Sorted List, Adding Items to a, 5-19
- Sorting by Multiple Keys, 5-20
- Sorting by Substrings, 5-18
- Sorting to a Vector, 5-21
- Special-Character Images, 15-25
- Specifiers
 - Binary, 15-23
 - Character, 15-21
 - Digit, 15-15
 - Exponent, 15-15
 - Numeric, 16-17
 - Radix, 15-15
 - Repeatable, 15-29
 - Sign, 15-15
 - Special-Character, 15-25
 - Termination, 15-27
- Specifiers, Additional Image, 10-11
- Specifiers, Numeric Image, 10-8
- Specifier, Subarray, 4-14
- Specifying an I/O resource, 14-20
- Speed, Execution, 14-29
- Speed, pen, 9-8
- SPOLL function, 21-8, 21-14
- Spooler Directories, Writing Files to the, 10-13
- Spooler directory, 10-12

- Spoolers, plotter, 9-12
- Spooler, Using a, 10-12
- Spooler, Using SRM Printers through the, 10-12
- Spooling, 10-12
- Spooling (SRM) Using PRINTER IS, 10-13
- SQRT function, 3-13, 3-19
- SRM binary, 11-2
- SRM file, 7-13
- SRM plotter spoolers, 9-12
- SRM Printers through the Spooler, Using, 10-12
- start bit
 - serial I/O, 21-27
- Statements
 - AREA COLOR, 8-54, 8-56
 - AREA INTENSITY, 8-48, 8-54, 8-56
 - AREA PEN, 8-54, 8-56
 - AXES, 8-13
 - BINCMP, 9-6
 - CLIP, 8-38
 - CLIP OFF, 8-9, 8-31
 - CONTROL KBD, 8-64
 - CSIZE, 8-19
 - DEG, 8-24
 - DUMP DEVICE IS, 9-2
 - DUMP GRAPHICS, 9-2
 - GESCAPE, 8-43, 8-59
 - GINIT, 8-4
 - GLOAD, 8-44
 - GRAPHICS ON, 8-4
 - GSEND, 9-8, 9-11
 - GSTORE, 8-43
 - IDRAW, 8-53
 - IMOVE, 8-53
 - IPLOT, 8-48, 8-53
 - LABEL, 8-11, 8-19, 8-26
 - LDIR, 8-24
 - LINE TYPE, 8-42
 - LORG, 8-23
 - MERGE ALPHA WITH GRAPHICS, 8-62
 - MOVE, 8-11
 - OUTPUT, 9-3
 - PDIR, 8-49, 8-54
 - PIVOT, 8-49, 8-54
 - PLOT, 8-3, 8-11, 8-45, 8-48
 - PLOTTER IS, 9-8
 - POLYGON, 8-54
 - POLYLINE, 8-55
 - RAD, 8-24
 - RATIO, 8-16
 - RECTANGLE, 8-55
 - RND, 8-4, 8-6
 - RPLOT, 8-49
 - SEPARATE ALPHA FROM GRAPHICS, 8-62
 - SHOW, 8-5, 8-16
 - SYMBOL, 8-48, 8-56
 - VIEWPORT, 8-9-10, 8-16, 8-38
 - WINDOW, 8-16, 8-21
- Statement-Termination Modifiers, 16-24
- STATUS statement, 5-28, 17-2, 21-20, 21-30
- STEP** key, 13-10
- stop bits
 - serial I/O, 21-27, 21-33
- STOP statement, 2-4
- Storage and Retrieval of Arrays, 7-5
- Storage Format for INTEGER Variables, 3-9
- Storage Format for REAL Variables, 3-9
- STORE statement, 6-23, 7-1
- Storing Data in Programs, 7-1
- Storing Data in Variables, 7-2
- String, 5-1
- String Array, Cataloging to a, 7-19
- String Arrays, 5-3
- String Arrays and Subarrays, Copying, 5-16
- String Arrays, Searching, 5-22

- String Concatenation, 5-4
- String Data, Entering, 16-8
- String, default dimensioned length of a, 5-2
- String Format, Standard, 15-3
- String Function, CHR\$, 5-13, 5-29
- String Function, DATE\$, 11-3
- String Function, DVAL\$, 5-23
- String Function, ERRM\$, 12-8, 12-14
- String Function, IVAL\$, 5-23
- String Function, LWC\$, 5-16, 5-24
- String Function, REV\$, 5-14
- String Function, RPT\$, 5-15
- String Functions, 5-14
- String Function, SYSTEM\$, 5-26
- String Function, TIME\$, 11-3, 11-6
- String Function, TRIM\$, 5-15
- String Function, UPC\$, 5-16, 5-24
- String Function, VAL\$, 5-12
- String images, 15-21, 16-19
- String Image Specifiers, 10-10
- String Length, Current, 5-9
- String Length, Maximum, 5-10
- String Manipulation, 5-1
- String-Related Functions, 5-9
- String Repeat, 5-15
- String Reverse, 5-14
- Strings, Searching for, 5-22
- String Storage, 5-2
- String-to-Numeric Conversion, 5-11
- String, Trimming a, 5-15
- String variable, 5-1
- String-variable names, 14-20
- Subarray, Copying a Subarray into another, 4-18
- Subarrays, Copying, 4-13
- Subarrays, Copying String Arrays and, 5-16
- Subarray Specifier, 4-14
- Subarray specifier examples, 4-15
- Subarrays, Rules for Copying, 4-20
- SUBEND statement, 6-24, 6-26
- SUBEXIT statement, 6-27
- Subprogram, 2-7
- Subprogram and User-Defined Function Names, 6-5
- Subprogram, Calling and Executing a, 6-3
- Subprogram, Difference Between a User-Defined Function and a, 6-5
- Subprogram Libraries, Using, 6-20
- Subprogram Location, 6-4
- Subprograms, A Closer Look at, 6-3
- Subprograms and Softkeys, 6-17
- Subprograms and Subroutines, Differences Between, 6-4
- Subprograms and the RECOVER Statement, 6-18
- Subprograms and User-Defined Functions, 6-1
- Subprograms at Once, Loading Several, 6-22
- Subprograms, Calling, 13-4
- Subprograms, Deleting, 6-23, 6-26
- Subprograms from the Keyboard, Calling, 6-20
- Subprograms in a PROG File, Listing the, 6-20
- Subprograms, Inserting, 6-25
- Subprograms Libraries, 6-20
- Subprograms, Loading, 6-21
- Subprograms, Merging, 6-26
- Subprograms One at a Time, Loading, 6-21
- Subprograms Prior to Execution, Loading, 6-22
- Subroutine, 2-7
- Subscript expression, 4-15
- Subscript range, 4-15
- SUB statement, 6-3, 6-6, 6-25
- Substring Position, 5-10
- Substrings, 5-5

- Substrings, Double-Subscript, 5-6
- Substrings, Single-Subscript, 5-5
- Substrings, Sorting by, 5-18
- SUM function, 3-14
- Suppressing the Catalog Header, 7-21
- Suspended Transfers, 20-30
- Symbol coordinate system, 8-22, 8-57
- Symbol program, 8-57
- SYMBOL statement, 8-48, 8-56
- system controller
 - description, 21-4
 - HP-IB status register, 21-15
- System printer, 10-1
- SYSTEM PRIORITY statement, 18-8
- SYSTEM\$ string function, 5-26

- T**
- TAN function, 3-15
- TANH function, 3-15
- Terminating a Transfer, 20-20
- Terminating Enters that Use Images, 16-22
- Termination Conditions, Default, 16-23
- Termination Conditions, Multiple, 20-17
- Termination, HPGL command, 9-8
- Termination specifier, 15-27
- Terminology, 14-1
- Tick marks, 8-13
- Time and Date Functions, 3-22
- TIMEDATE function, 3-22, 11-3, 11-6
- Time Format, Clock, 11-4
- TIME function, 3-22, 11-6
- Time of Day, 11-16
- Timeout Events, Setting Up, 18-22
- Timeout limitations, 18-22
- timeouts
 - GPIO interface, 21-43
- Timeouts, Interface, 18-21
- Timeouts, Interrupts and, 18-1
- Time, Setting Only the, 11-6
- TIME\$ string function, 11-3, 11-6

- Timing Compatibility, 14-5
- Timing, Interval, 11-13
- Titles, 8-11, 8-19
- Top-Down Design, 6-28
- TRACE ALL statement, 13-11
- TRACE OFF statement, 13-15
- TRACE PAUSE statement, 13-14
- Tracing, 13-11
- TRANS binary, 20-1, 21-1, 21-36
- Transfer Event-Initiated Branching, 20-19
- Transfer examples, 20-22
- Transfer Formatting, 20-14
- TRANSFER in BASIC/UX, 20-36
- TRANSFER, locking an interface, 20-36
- Transfer methods
 - BASIC/DOS, 20-13
 - BASIC/UX, 20-13
 - BASIC/WS, 20-13
- Transfer Methods
 - BASIC/UX, 20-37
- Transfer methods and rates, 20-34
- Transfer parameters, 20-14
- Transfer performance, 20-30
- Transfer rates, 20-35
- TRANSFER Records and Termination, 20-18
- Transfer restrictions, 20-3, 20-39
- Transferring a Specified Number of Bytes, 20-16
- Transferring Records, 20-17
- Transfers
 - Burst with BASIC/UX, 20-13
 - DMA with BASIC/UX, 20-13
- Transfers and Disk Drives, Overlapped, 20-31
- Transfers, Continuous Non-Overlapped, 20-16
- Transfers, Inbound and Outbound, 20-2
- Transfers Indefinitely, Continuing, 20-15
- Transfers, Non-Overlapped, 20-15

- Transfer Sources and Destinations, Supported, 20-3
 - Transfers, Suspended, 20-30
 - TRANSFER statement, 20-1, 20-10-12, 20-40
 - Transfer status, 20-14
 - Transfers, The Purpose of, 20-1
 - Transfer techniques, 20-1
 - Transfer, Termination, 20-20
 - Transfer Termination, 20-14
 - TRANSFER termination, premature, 20-42
 - Transfer types, 20-34
 - Translating a drawing, 8-49
 - Trapping and Recovery, Scope of Error, 12-6
 - Trapping Errors with BASIC Programs, 12-5
 - Trapping HP-UX signals from BASIC/UX, 18-23
 - Trapping (OFF ERROR), Disabling Error, 12-6
 - TRIGGER statement, 21-8, 21-10
 - Trigonometric Functions, 3-14
 - Trigonometric Mode, COMPLEX Arguments and the, 3-20
 - Trimming a String, 5-15
 - TRIM\$ string function, 5-15
 - Turning on Graphics, 8-4
 - Two-dimensional, 4-2
 - Two-Dimensional REAL Array, 4-4
 - Type Conversions, Implicit, 3-4
 - Type, Line, 8-41
 - Types of Events, 18-1
- U**
- UART, 21-26
 - UDC, 8-57
 - UDUs, 8-5, 8-9, 8-16
 - Unequal units (anisotropic scaling), 8-7
 - Units
 - Anisotropic, 8-7
 - Equal (isotropic scaling), 8-5, 8-16
 - GDUs, 8-9, 8-16
 - Graphics display, 8-9, 8-16
 - Isotropic, 8-5, 8-16
 - UDUs, 8-9, 8-16
 - Unequal (anisotropic scaling), 8-7
 - User-defined (UDUs), 8-5, 8-9, 8-16
- Unnamed buffer, 20-6
- Unnamed Buffers, Assigning I/O Path Names to, 20-7
- Unreferenced entries, 13-8
- Unused Entries, 13-7
- UPC\$ string function, 5-16, 5-24
- User-defined characters, 8-56
- User-defined units, 8-5, 8-16
- V**
- VAL function, 5-11
 - VAL\$ string function, 5-12
 - Value, Pass by, 6-7
 - Variable Initialization, 6-17
 - Variable names (syntax of), 14-25
 - Variables, Assigning, 3-4
 - Variables, Declaring, 3-3
 - Variables, naming, 3-3
 - Variables, numeric, 3-2
 - Variables, Using Program, 13-2
 - Vector, Sorting to a, 5-21
 - Video, inverse, 9-6
 - VIEWPORT statement, 8-9-10, 8-16, 8-38
 - Volatile real-time clocks, 11-2
 - VS (velocity select) command, 9-8
- W**
- WAIT FOR statement, 20-20
 - WAIT parameter, 20-20
 - WAIT statement, 2-5
 - Week, Day of the, 11-13
 - WHILE ... END structure, 2-15

WHILE . . . END WHILE structure, 2-18,
2-20
WHILE statement, 2-18
WINDOW statement, 8-16, 8-21
WORD attribute, 19-6-7, 19-11
Write-enable mask, 8-59
WRITEIO statement, 18-18
Writing Files to the Spooler Directories,
10-13

X

X-axis scaling, 8-6, 8-16
Xmax, 8-16
XREF statement, 13-6

Y

Y-axis scaling, 8-7, 8-16
Ymax, 8-16



**HEWLETT
PACKARD**

**HP Part Number
98616-90010**

Printed in U.S.A. E0691



98616-90623 Manufacturing Number