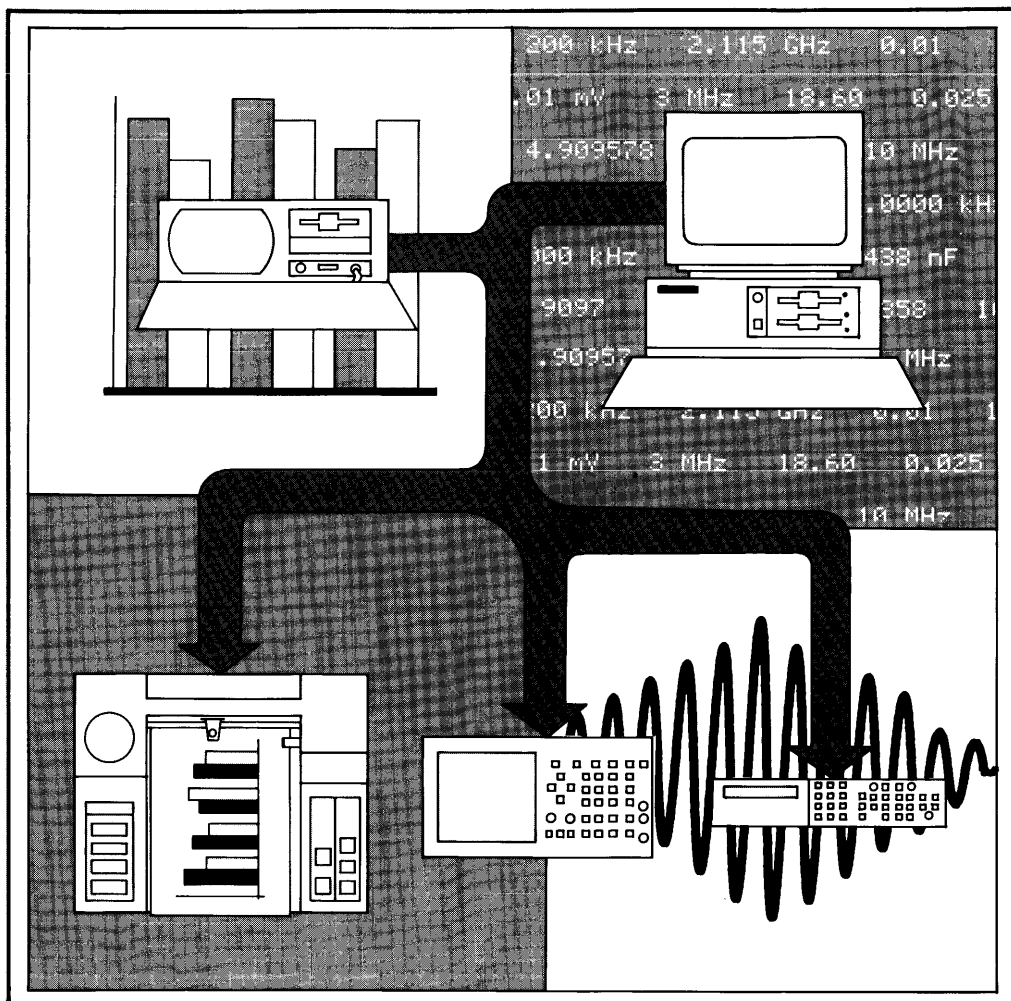


HEWLETT-PACKARD



Using HP BASIC for Instrument Control

A Self-Study Course

Notice

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

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

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

©1988 by Hewlett-Packard Co.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

Corvallis Workstation Operation
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Using HP BASIC for Instrument Control

A Self-Study Course (Volume 1)



Edition 1 October 1988

82302-90001

Printing History

Edition 1

October 1988 Mfg. No. 82302-90001

Contents

Introduction

About This Course	1
What You'll Need	2
Where to Begin the Course	3
Before You Begin	4
Use BASIC Keyboard Overlays!	4
Load Those Binaries!	4
Learning About Your Computer	5
Calling All PC Users...	7

Part 1: Basic BASIC Programming

Lesson 1 The Display and Keyboard

Using the Keyboard	1-2
The [ENTER] Key	1-2
[SHIFT] and [CAPS LOCK]	1-4
Number Keypad	1-4
[CTRL]	1-5
[Alt]	1-5
Softkeys	1-5
The Display	1-8
Output Area	1-8
Display Line	1-10
Keyboard Area	1-10
Message and Results Line	1-11
Softkey Labels	1-11

Erasing What You Don't Want	1-12
CLEAR LINE	1-13
CLEAR SCREEN	1-13
KEY LABELS ON/OFF	1-14
Math Functions	1-14
Keeping a Trail of Your Work	1-14
Addition	1-15
Subtraction	1-15
Multiplication	1-16
Division	1-16
Raising a Number to a Power	1-17
Exponents and Powers of 10	1-17
Other Math Functions	1-18
What About Longer Expressions?	1-21
Fun Functions	1-23
Review Quiz	1-25

Lesson 2 Your First Program

Solving a Problem in Calculator Mode	2-2
Writing a Program	2-3
What Is a Program?	2-3
Edit Mode	2-3
Line Numbers	2-4
Running a Program	2-7
RUN	2-7
If the Program Doesn't Work	2-9
Parts of the Program	2-9
Statements and Functions vs. Commands	2-10
Using the [STEP] Key	2-11
INPUT	2-11
Variables	2-11
LET	2-12
The DISP Statement	2-13
The All-Important END	2-14

Editing Your Program	2-15
Editing a Line	2-15
Entering New Lines	2-17
Moving Program Lines	2-18
Renumbering Lines	2-19
Run the Edited Program	2-20
Clearing the Screen	2-21
Waiting, Waiting	2-21
A Friendlier Display	2-22
Review Quiz	2-23

Lesson 3 Saving Your Program

Write a New Program	3-2
Run the Program	3-3
[PAUSE] and [CONTINUE]	3-3
The Run Light	3-4
REMARKS	3-5
Telling Output Where to Go	3-6
The FOR-NEXT Loop	3-8
PRINT vs. DISP	3-9
Comma vs. Semicolon	3-10
Getting Out of Edit Mode	3-10
Listing Your Program	3-10
Partial Listing	3-11
What About LIST BIN?	3-11
Making Your Program Permanent	3-14
Specifying Mass Storage Is	3-15
Initialize a Disk	3-17
Format and Interleave Factor	3-19
About Disks	3-19
Storing and Loading	3-22
Re-Storing the Program	3-23
What About File Names?	3-23
Scratching a Program from Memory	3-23
Loading the Program	3-24
Saving and Getting a Program	3-25

Seeing a Catalog	3-26
Directing Output	3-27
Purging a Program from Mass Storage	3-27
Review Quiz	3-27

Lesson 4 Handling Numbers

Pre-Run and Run	4-2
Variables	4-2
Variable Names—How Long?	4-3
Types of Variables	4-3
Declaring Variable Types	4-4
Assigning Numbers to Variables in a Program	4-6
The Dynamic Duo of DATA and READ	4-6
The Data Stream	4-7
The Data Pointer	4-9
Restoring the Data Pointer	4-9
Assigning Numbers from the Keyboard	4-10
Declaring a Variable	4-13
Rounding a Number	4-14
Be Careful With INTEGER	4-15
Review Quiz	4-15

Lesson 5 Handling Words in Strings

What Is a String?	5-2
String Variables	5-2
The String and the Variable	5-3
Assigning Strings to Variables	5-4
Reserving Memory	5-5
INPUT and LINPUT	5-8

Fun With Strings	5-9
The "Null String"	5-9
Replacing a String	5-10
Putting Words Together	5-10
Reversing a String	5-11
Using Parts of Strings	5-12
Replacing Part of a String	5-15
Finding the Length of a String	5-16
Finding Position Within a String	5-16
Strings, Semicolons, and Spacing	5-17
Conversions	5-19
Converting Strings to Numbers	5-19
Characters to Numbers	5-20
Numbers to Characters	5-21
Inserting Quotation Marks	5-21
Lowercase and Uppercase Conversions	5-22
Useful String Functions	5-23
Review Quiz	5-25

Lesson 6 Decisions, Decisions

The GOTO Statement	6-1
Changed Line Numbers	6-3
Line Labels	6-3
Forget GOTO!	6-4
Subroutines	6-4
Subprograms	6-7
Subprogram Components	6-8
Why Use Subprograms?	6-9
Making Decisions	6-10
IF-THEN	6-10
Comparisons	6-11
Be Careful With Comparisons	6-12
IF-THEN with AND-OR	6-13
IF-THEN with END IF	6-15
IF-THEN with END IF and ELSE	6-15
SELECT-CASE	6-27
Review Quiz	6-29

Lesson 7 Repetition, Repetition

The FOR-NEXT Loop	7-1
Specifying a STEP	7-2
Negative Step	7-3
REPEAT and UNTIL	7-4
Indenting to Taste	7-7
WHILE-END	7-8
LOOP-END	7-9
Live Keyboard	7-11
Functions	7-13
Local Variables	7-15
Functions vs. Subprograms	7-16
Review Quiz	7-16
Laboratory Exercise	7-17

Lesson 8 The Marvelous Array

What Is an Array?	8-2
Subscripted Variables	8-3
Putting Data into an Array	8-4
Setting the Base Element	8-4
Dimensioning the Array	8-5
Putting Data into the Array	8-9
Using a Loop	8-9
Using the Asterisk	8-11
Using Elements of an Array	8-11
Strings in Arrays	8-12
Special Array Functions	8-18
Finding the Dimensions	8-18
Finding Out the Option Base	8-19
Finding Out the Data	8-20
Summing the Array	8-21
Structured Programming	8-23
Review Quiz	8-24

Lesson 9 Printing to Please

Formatting the Easy Way	9-1
More Sophisticated Formatting	9-4
Using PRINT USING	9-5
Printing Numbers	9-6
Multiple Images in One Statement	9-9
Printing Strings	9-11
Other Uses for PRINT USING	9-13
Using an Image	9-15
Review Quiz	9-17
Laboratory Exercise	9-20

Lesson 10 Using Mass Storage

A Data Storage Example	10-2
Reviewing Program Storage	10-3
Specifying Mass Storage	10-3
Initializing a Disk	10-3
Storing a Program	10-4
Adding a Label	10-4
Programs vs. Data	10-5
Create a Data File	10-5
All About Files	10-6
What Kind of File to Use?	10-7
How Large a File?	10-8
The CREATE Statement	10-10
Open a Path to the File	10-13
Output Data Along the Path	10-14
Serial Access	10-14
Random Access	10-14
Close the Path to the File	10-15
Entering Data from Disk	10-16
A Random Example	10-20
Review Quiz	10-23
Laboratory Exercise	10-24

Part 2: Instrument Control with HP-IB

Lesson 11 Introduction to HP-IB

What Is an Interface?	11-3
Interfaces in Your Computer	11-3
Compatibility: Four Vital Areas	11-5
The HP-IB	11-5
Computer Bits and Bytes	11-5
HP-IB Features	11-7
What's the Difference?	11-7
On the Bus	11-8
Roles on HP-IB	11-9
What a Device Can Do	11-10
The Bus Lines	11-12
Data Lines	11-12
Handshake Lines	11-13
Bus Management	11-14
Review Quiz	11-16

Lesson 12 Installing HP-IB Hardware

Identifying HP-IB Devices	12-1
Pins on the Connector	12-2
Inside the "HP-IB-Capable" Device	12-3
Determining What an Instrument Can Do	12-3
Setting Addresses	12-8
Interface Select Code	12-9
Device Address	12-10
Reading a Device Address	12-10
Secondary Addresses	12-12
Changing a Device Address	12-12
What Cables to Use	12-15
How to Connect Devices	12-16
What About Cable Length?	12-18
Keep Those Instruments On!	12-18
Review Quiz	12-19

Lesson 13 Take Control of Those Instruments!

Addressing Instruments (A Quick Review)	13-1
Using a Name	13-2
Using a Path	13-3
Calling Instruments to Attention	13-3
Using ABORT	13-4
The REMOTE Statement	13-5
Going Back to LOCAL	13-6
Taking Control—And Keeping It	13-6
The LOCAL Statement	13-7
Using CLEAR	13-8
A Programmed Example	13-11
Entering the Program	13-11
Running the Program	13-12
Review Quiz	13-13

Lesson 14 Telling Instruments What to Do

What the Instrument Needs	14-1
A Manual Example	14-2
Doing It With HP BASIC	14-5
The OUTPUT Statement	14-5
What OUTPUT Does	14-7
Instrument Commands	14-7
Where to Find Instrument Commands	14-9
Specifying the Instrument Command	14-9
An HP-IB Example	14-10
Set the Voltage	14-10
Vary the Output	14-11
Watch Those End Lines!	14-12
Review Quiz	14-13

Lesson 15 Getting Information from an Instrument

A Manual Example	15-1
Using ENTER	15-2
What ENTER Does	15-3
Specifying the Device	15-3
Specifying the Variable	15-5
OUTPUT and ENTER: Hand-in-Hand	15-8
Triggering an Instrument	15-10
The Instrument's Trigger Command	15-11
The TRIGGER Statement	15-11
Instruments Are Smart Too!	15-14
Review Quiz	15-17

Lesson 16 How an Instrument Summons Service

Requesting Service	16-2
Reading Registers with STATUS	16-3
HP-IB Status Registers	16-5
When an SRQ Is Issued	16-7
How to Detect the SRQ	16-12
Detecting a Status Change	16-12
Using an Interrupt	16-15
Enabling Other Interrupts	16-19
Do You Need to Mask the SRQ?	16-20
The Status Byte Tells All	16-20
The Status Byte	16-20
Unmasking the Status Byte	16-22
The Beauty of Bit 6	16-23
Reading the Status Byte	16-23
No Masking Necessary	16-29
Review Quiz	16-29

Lesson 17 Saving and Reusing Instrument Data

What Is Instrument Data?	17-2
Data Formats	17-2
Data Files	17-3
Data OUTPUT	17-3
ENTERed Data	17-4
The Number Builder	17-5
Storing Instrument Data	17-11
Retrieving Data	17-16
Using OUTPUT USING	17-17
Review Quiz	17-19
Laboratory Exercise	17-20

Lesson 18 Making the Keyboard Work for You

How to Use Softkeys	18-2
Softkeys as Typing Aids	18-3
Listing the Softkeys	18-3
Rewriting a Softkey Definition	18-4
Changing a Key from a Program	18-8
Common System Keys	18-11
Storing Softkey Definitions	18-11
Erasing Softkey Definitions	18-12
Loading Softkeys	18-12
Keys for Program Branches	18-17
Using ON KBD	18-21
Review Quiz	18-21

Lesson 19 Data on Display

The Graphics Display	19-2
Showing the Graphics Plane	19-2
Locations in the Graphics Area	19-3

Graphics Fundamentals	19-5
Initializing Graphics	19-5
Clearing Graphics	19-5
Drawing with the Pen	19-6
Choosing Pen Type	19-6
Using PLOT	19-8
Choosing Line Type	19-10
Turning Graphics OFF	19-11
Showing the Alpha Plane	19-11
A Typical Graphics Application	19-12
Know Your Instrument	19-13
Initialize the Graphics Plane	19-13
Specify the Plotter	19-13
Turn Graphics On	19-14
Clear and Initialize	19-14
Use FRAME	19-14
Set the Viewport	19-14
Using VIEWPORT	19-15
A Different VIEWPORT	19-16
Scale the Plotting Area	19-18
Put in Axes Lines	19-22
More AXES Control	19-25
Plotting the Trigolator Axes	19-26
Put in a Grid	19-26
Label Your Plot	19-29
Clipping and Unclipping	19-30
Using LORG	19-32
Using CSIZE	19-33
Using LDIR	19-34
Plot Your Data	19-36
Dump to a Printer	19-42
There's a Lot More...	19-42
Review Quiz	19-43
Laboratory Exercise	19-46

Lesson 20 How to Design a Complete Program

Writing a Structured Program	20-2
Determine a Manual Solution	20-5
Check Out Hardware	20-6
Create a Warnier-Orr Diagram	20-6
Determine Subprograms and Subroutines	20-10
Write the Program Code	20-11
Test and Debug	20-19
Debugging Aids	20-19
Hints for Debugging	20-20
Document the Program	20-21
Review Quiz	20-22
Laboratory Exercise	20-24

Part 3: Increase Your Instrument Control

Lesson 21 Sending Custom Bus Messages

Levels of Control	21-3
Bus Messages	21-4
Using SEND	21-5
The Bus Messages	21-8
A Few Points...	21-11
Sending Bus Messages	21-12
Message Mnemonics	21-13
Some Common Bus Activities	21-16
How to Send a Bus Message	21-19
"Hands-On" Experience	21-20
Simulating an OUTPUT Statement	21-20
Simulating an ENTER Statement	21-21
Simulating SPOLL	21-21
Review Quiz	21-28

Lesson 22 Multiple Instruments, Multiple Controllers

Addressing Multiple Instruments	22-1
Multiple Listeners	22-2
Secondary Addressing	22-3
The Parallel Poll	22-5
Configuring Instruments to Respond	22-6
Conducting a Parallel Poll	22-7
Using PPOLL UNCONFIGURE	22-9
More Than One Controller	22-10
Passing Control	22-10
Interrupts While Non-Active Controller	22-12
Addressing a Non-Active Controller	22-17
Requesting Service	22-18
PPOLL RESPONSE	22-19
Review Quiz	22-21

Lesson 23 Controlling Register Contents

Registers	23-1
Reading a Status Register	23-3
Changing a Control Register	23-5
Using the CONTROL Statement	23-5
READIO and WRITEIO Registers	23-6
Review Quiz	23-8

Lesson 24 Subprogramming

Calling and Executing a Subprogram	24-2
A Word about Context	24-2
Calling Subprograms from the Keyboard	24-3

Passing Parameters	24-3
Passing By Value	24-4
Passing By Reference	24-4
Parameter Lists	24-9
OPTIONAL Parameters	24-10
Using NPAR	24-10
Using Common Blocks	24-13
COM vs. Pass Parameters	24-17
Hints for Using COM Blocks	24-18
A Practical Example	24-22
Deleting a Subroutine	24-25
Review Quiz	24-26
Laboratory Exercise	24-28

Lesson 25 Output Data in Different Formats

Types of OUTPUT	25-2
Free-Field OUTPUT	25-2
Separators and Terminators	25-3
Array Separators	25-4
Free-Field OUTPUT with END	25-7
END with OUTPUT to HP-IB	25-8
END with OUTPUT to Files	25-8
OUTPUT with Attributes ASSIGNEd	25-8
The ASSIGN Statement	25-9
Finding Information About I/O Paths	25-11
Assigning Attributes	25-12
Changing the EOL Sequence	25-15
OUTPUT Using Images	25-19
The OUTPUT USING Statement	25-20
Images	25-20
How the Computer Looks at an Image	25-21
Image Definitions During OUTPUTS	25-23
Additional Image Features	25-34
END with Images	25-36
END with HP-IB	25-37
Review Quiz	25-39

Lesson 26 Entering Data in Different Formats

Free-Field Enters	26-2
Item Separators	26-2
Item Terminators	26-2
Entering Numeric Data with the Number Builder	26-3
Entering String Data	26-9
Terminating ENTER Statements	26-11
EOI Termination	26-11
EOI with Numeric Characters	26-13
ENTER with Assigned Attributes	26-14
Using Files	26-14
Specifying Data Size with ASSIGN	26-15
ENTERS with Images	26-16
The ENTER USING Statement	26-16
Images	26-17
How an Image Is Used for ENTER	26-17
Numeric Images with ENTER	26-19
String Images	26-21
Ignoring Characters	26-23
Binary Images	26-24
Terminating ENTERs That Use Images	26-26
Redefining EOI	26-26
Changing Terminations for ENTER USING	26-28
Other Image Features	26-31
Using Instrument Data Formats	26-35
ASCII Data Example	26-35
REAL Data Example	26-38
Using the Instrument's Internal Format	26-42
The Instrument Learn String	26-46
Review Quiz	26-49
Laboratory Exercise	26-50

Lesson 27	Buffers and Buffered I/O	
	The TRANSFER Statement	27-1
	How TRANSFER Works	27-4
	TRANSFER Parameters	27-4
	When Can You Use TRANSFER?	27-6
	Buffers	27-6
	Types of Buffers	27-8
	Buffer Registers	27-11
	Buffer Pointers	27-11
	A Real-World Example	27-19
	Review Quiz	27-22

Lesson 28	All About Interrupts	
	Using Interrupts	28-2
	What Happens in an Interrupt?	28-2
	Using RECOVER	28-2
	How Many Interrupts?	28-3
	Real-Time Interrupts	28-7
	Timeouts	28-8
	Data Transfer Interrupts	28-9
	Error Trapping	28-9
	The Mouse	28-11
	External HP-IB Interrupts	28-16
	What an Interrupt Needs	28-20
	Setting Up a Branch	28-20
	Enabling the Event	28-20
	Logging the Event	28-21
	Software Priority	28-21
	Hardware Priority	28-27
	Review Quiz	28-29

Lesson 29 Streamline Your Programs

Slashing Space Requirements	29-2
Use BDAT Files	29-2
Reduce Remarks	29-3
Boosting Speed	29-3
Benchmarking	29-3
Use Look-Up Tables	29-7
Use Integers...Sometimes	29-8
Subprograms and Speed	29-9
Use COM	29-11
Hardware Improvements	29-11
Review Quiz	29-12
Laboratory Exercise	29-12

Lesson 30 Tricks and Techniques

Simulating a Key	30-1
Sneaking Past Errors	30-2
Extending Control	30-3
Use Subprogram Libraries	30-5
Loading Subprograms	30-5
Loading Subprograms One at a Time	30-6
Loading Several Subprograms at Once	30-6
Deleting Subprograms Automatically	30-8
Use Passthrough Mode	30-13
Review Quiz	30-16

Appendixes and Index Appendix A: Answers to Review Questions

Appendix B: IEEE-488 Interface Capability Codes

Appendix C: HP-IB Status and Control Registers

Appendix D: HP-IB Bus Messages

Index

Sidebars

The Trouble Killers	8
Whither QWERTY?	1-6
A Bit of BASIC History	2-6
If You Have Printing Problems...	3-12
Initializing a Disk on a PC	3-21
Five Fail-Safe Rules for Variables	4-5
Figurative Flowcharts	6-22
The Bubble Sort	8-15
Of Files HP-UX and DOS	10-12
HP-IB: A Short History	11-2
HP-IB Addresses and Switch Settings	12-13
To Mask or Unmask?	16-27
Editing Softkeys on a PC	18-13
Self-Computing the Scale	19-40
HP-IB Handshaking	21-25

Featured HP Instruments

The HP 3326A Two-Channel Synthesizer	13-9
The HP 6624A Multiple Output Power Supply	14-3
The HP 438A Dual Sensor Power Meter	15-6
The HP 8753 Network Analyzer	16-8
The HP 8590A Portable RF Spectrum Analyzer	17-7
The HP 8980A Vector Analyzer	18-14
The HP 3325B Synthesizer/Function Generator	20-3
The HP 3456A Digital Multimeter	20-4
The HP 8340B Synthesized Sweeper	21-2
The HP 8720A Microwave Network Analyzer	22-4
The HP 6030A Autoranging System Power Supply	24-20
The HP 8510B Network Analyzer	26-32
The HP 3457A Multimeter	27-18
The HP 3852A Data Acquisition and Control System	28-13
The HP 8757A Scalar Network Analyzer	30-10
The HP 8350B Sweep Oscillator	30-12

Introduction

Welcome to *Using HP BASIC for Instrument Control*. This self-study course consists of 30 lessons, in three parts. You can start with part 1, part 2, or part 3, depending upon your abilities and experience.

The course is designed to teach you how to use a computer to control electronic instruments. The computer must be running the HP BASIC language, and most of the control operations you'll learn require the interface known as HP-IB (or IEEE-488).

About This Course

Part 1 of the course is an introduction to programming in HP BASIC.

Part 2 explains HP-IB and simple instrument control. By the time you've finished this part, you should be able to control most functions of Hewlett-Packard instruments with HP-IB.

Part 3 shows you more elegant and powerful programming for instrument control. This part will help you speed up your programs, and you will learn techniques to control instruments with non-standard formats, whether those instruments are from HP or other manufacturers.

The HP BASIC Instrument Control Examples disk contains examples, laboratory exercises, and utility programs.

What You'll Need

To work through the examples in this course, you should have one of the following systems:

- An HP 9000 Series 200 or 300 computer with HP BASIC.
- An HP PC-305 or PC-308 HP BASIC Controller.
- An HP Vectra personal computer with the HP BASIC Language Processor card installed.
- An IBM AT or AT-compatible personal computer with the HP BASIC Language Processor installed.

Most of the examples in part 1 can be done with *any* version of HP BASIC. Newer versions (3.0 and up) have a greater number of statements created specifically for instrument control, so you'll be better off using one of these for parts 2 and 3.

You don't need HP-IB or instruments to do any of the examples in part 1 of this course. Parts 2 and 3 will certainly be more interesting if you can try some of the examples with actual electronic test instruments connected to your computer via HP-IB. But if you don't have any interface cables or HP-IB instruments yet, you can learn a lot just by looking closely at the program code. Many of the examples are themselves actual "real-world" applications, so they'll be a good reference later when you do bring your HP-IB instrumentation on line.

This course doesn't cover *all* of HP BASIC—it concentrates on the most often-used features and on techniques you'll need specifically for instrument control. Your other HP BASIC documentation is the place to look for further details and for a full list of statements and commands.

Where to Begin the Course

Depending on your experience and knowledge of BASIC in general and HP BASIC in particular, you may be able to jump into part 2 or part 3, or to skip around without doing lessons in order.

New to programming? If you're a brand-new programmer in BASIC, read the rest of this introduction carefully. Then start with part 1 (lessons 1-10) and work through it carefully. Do the examples and the review quizzes. Be sure you understand everything in part 1 before you go on to part 2 and part 3.

Memory hazy, or new to this version of BASIC? If it's been a while since you've used HP BASIC, or you're new to the latest version, start with part 1 and skim over the material you already know. HP BASIC does have some bells and whistles you haven't discovered yet—features you'll appreciate when you begin to write code. Read the rest of this introduction, skim part 1 quickly, then go on to part 2 (lessons 11-20).

Already comfortable with HP BASIC? Then finish this introduction and dive right into part 2—you're ready to learn how to handle instrumentation using the Hewlett-Packard Interface Bus. You'll find yourself using a lot of old familiar HP BASIC statements, as well as some new ones created specifically for instruments.

Already using HP-IB? Perhaps you've gotten your instruments working, but want to do more. Part 3 (lessons 21-30) is for you. In it, you'll learn tricks and techniques to increase control and make your program smaller and faster. Read this introduction, then go right to the beginning of part 3.

No matter where you start, you'll find each lesson is self-contained and "bite-sized" for easy digestion. Work through a lesson today, a lesson tomorrow, a lesson next week. Skip around if you like. In no time at all, you'll be writing sophisticated control programs.

Before You Begin

Before you start this course, you should be able to turn on your computer and load BASIC — that is, get the HP BASIC screen on your computer's monitor. If you don't know how to do this, refer to the manual for your computer (or the HP BASIC Language Processor card) or get someone to help you.

Use BASIC Keyboard Overlays!

Because this course covers all HP computers that can run HP BASIC (as well as the HP BASIC Language Processor for a Vectra or other personal computer), you won't be shown the location of every key on your computer's keyboard.

This won't be a problem. Either your computer is already clearly labeled for BASIC, or you can affix a handy keyboard overlay that shows which keys to press on your computer. HP BASIC and the Language Processor come with these overlays, so you can turn whatever computer you have into a "dedicated" BASIC machine.

And if you don't see a particular key on the keyboard or overlay? Again, *no problem*: you can execute any command you need just by typing it in, character by character. Naturally, if you see a "shortcut" or a key that does something in one keystroke that it takes you 10 to type, by all means use it!

Load Those Binaries!

One other thing: HP BASIC has a number of binary programs available that add functions to the language.

When BASIC is first put into your computer, whoever installed it had the option of loading these programs along with it. Most people load the binaries as a matter of course. And if you have a choice (and enough memory) you should make sure they're *all* loaded before you begin this course.

If you don't (or can't) load them all at once, you'll get error messages when you try to run some of the examples. Then you'll have to stop to load the required binary program before proceeding.

Learning About Your Computer

There are times in this course when you need to know things about your computer. The SYSTEM_EX program on your disk of examples is a fast, easy way to see what your current configuration is.

Note



If you get an error message during the procedure below, look at lesson 3, Saving Your Program, for a more detailed explanation of how to use disks and disk drives.

Here's how to use the disk:

1. If you loaded BASIC from a floppy disk, insert the disk of examples *in the same disk drive from which you loaded BASIC*. Be sure to insert the disk all the way in. Close the drive door or turn the latch if there is one. Then go on to step 3.
2. If you loaded BASIC from a hard disk, you'll need to specify a floppy disk drive as the current "MSI" device. If you are using an HP Vectra PC or other personal computer with the HP BASIC Language Processor (or a PC-305 or PC-308), type:

```
MSI " : , 1500 , 0 "
```

If you are using an HP 9000 Series 200 or Series 300 computer, your disk drive may have one of several addresses. Typically, you would type:

```
MSI " : , 700 , 0 "
```

Refer to your owner's manual for further information.

Then press the [ENTER] key, or whatever key ends a line on your computer.

Now insert the disk of examples in the computer's *top or left-hand* disk drive, and proceed with step 3 below.

3. Now type:

```
LOAD "SYSTEM_EX"
```

Then press [ENTER] (or [RETURN], or whatever key ends a line on your computer).

4. You should see an asterisk in the lower right corner of the screen for a moment as the program is copied into your computer's memory.

5. Now type:

```
RUN
```

Then press [ENTER].

You should see a display similar to this one:

```
SYSTEM_EX  PROG      10      256      2558      6-Apr-88  8:21
AVAILABLE MEMORY : 915558
CRT ID       : 6: B0H G 1
DUMP DEVICE  : 701
GRAPHICS INPUT : 0
PLOTTER      : 0
KEYBOARD LANGUAGE : ASCII
LEXICAL ORDER : ASCII
MASS MEMORY  : 0000000000000000
MASS STORAGE IS : :CS00, 1500, 2
MSI          : :CS00, 1500, 2
PRINTALL     : 1
PRINTER      : 1
SERIAL NUMBER : 111111111111
SYSTEM ID    : 9816
SYSTEM PRIORITY : 0
TRIG MODE    : RAD
BASIC VERSION : 5.0
```

-

EDIT	Continue	RUN	SCRATCH	LOAD ...	LOAD BIN	LIST BIN	RE-STORE
------	----------	-----	---------	----------	----------	----------	----------

The display gives information about your system – what BASIC version, what current printer, etc. If you can't understand it all now, don't worry; by the end of the course you'll know all about this display.

You can get the same information at any time with the `SYSTEM$` statement. Just type `SYSTEM$`, followed by what you want to see (in parentheses and quotation marks). Then press the `[ENTER]` key. Here's an example:

```
SYSTEM$ ("VERSION: BASIC")  
[ENTER]
```

This shows you the version of HP BASIC you're using now.

That's it! It's time to begin. Turn to the right part of the course for you and start programming! (Unless you're using an HP Vectra PC or other IBM AT-compatible personal computer. If you have one of these machines, then read on a little further.)

Calling All PC Users...

The HP BASIC Language Processor card can be used in virtually any IBM AT-compatible computer, such as the HP Vectra PC.

If you're working with a PC, you must use different key combinations to perform BASIC commands and statements. For instance, you'll sometimes need to press the `[CTRL]` key and another key, or the `[Alt]` key and another key.

You'll find these key combinations shown on the Language Processor keyboard overlays for your computer. And all key combinations are shown in the programmer's reference guide for the Language Processor card. You may want to copy the list and keep a copy at your work site for handy reference.

If you don't have an overlay, or if you can't remember a key, it's no problem; you can usually *type* the command or statement, letter by letter. Then press `[ENTER]` or `[RETURN]` to execute it.

The Trouble Killers

It happens to everyone – the computer "hangs," goes off into some number-crunching never-never land and doesn't return. Here's how to cure trouble without resorting to violence:

1. *Gentlest:* Press the [CLEAR LINE] key. This gently washes the offending line off the screen without affecting your program or anything else.
2. *Next gentlest:* CLEAR SCREEN (either press the key or type the statement). Again, this doesn't affect anything in the computer, just the screen.
3. *If a program is running, try:* The [CLEAR I/O] key followed by [STEP] or [CONT]. This suspends any current input and output and makes the program pause, then resume. It's a good solution for things that cause the computer to "hang" indefinitely – such as printing to a printer that isn't turned on.
4. *To scrub memory clean of all program lines:* SCRATCH or SCRATCH A clears out any program currently in memory. Don't worry – it can't affect mass storage, such as your disks or tape.
5. *When all else fails:* The [RESET] key stops a running program, clears all input/output, and generally takes charge. (It leaves your program still in memory, though.)
6. *Last resort:* The on-off switch is inelegant, but sometimes you'll just *have* to turn the computer off, then on again to get its attention. When you use this, you lose everything not stored on disk: that means you'll have to reboot BASIC, and if you had a program in the computer, it's gone, too.

Part 1

Basic BASIC Programming

This first part of the course consists of lessons 1-10. It teaches simple programming in HP BASIC, beginning with the most fundamental concepts.

You'll begin by learning about the various parts of the display and keyboard, but you'll soon be writing elementary programs. Here you learn essential techniques that you'll put to use later as your programming becomes more sophisticated.

Later in part 1 you'll learn how to handle numbers, words, and arrays of both. This part also teaches you how to change the format of printed output and how to store programs and data on disk.

If you're new to programming, or need a fast refresher course, start here. If you decide to skip right to part 2 (lessons 11-20) now, use the table of contents and index to refer back to part 1 if you hit something you don't understand.

Turn the page now to begin lesson 1.

The Display and Keyboard

In this section, you learn to use the HP BASIC language in a simple way, from the computer keyboard. This section begins at the very beginning, with the two parts of the computer you use most:

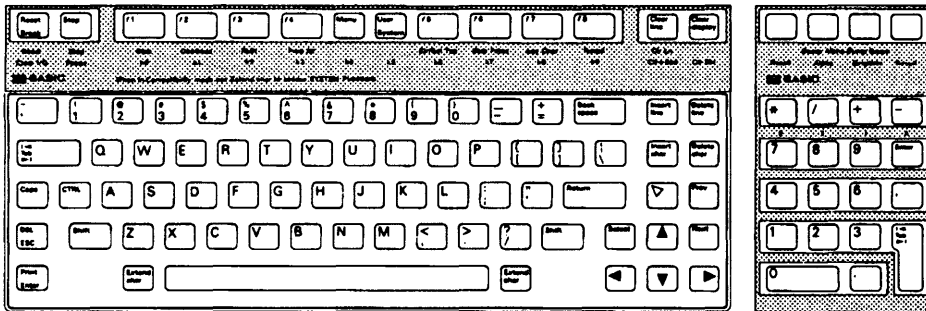
- The keyboard.
- The HP BASIC display.

To help you get comfortable with the computer and understand it better, you'll also learn a few simple HP BASIC commands:

- CLEAR LINE.
- CLEAR SCREEN.
- KEY LABELS.
- PRINTALL IS and [PRT ALL].
- Math functions and exponents of 10.
- The time, the date, and the BEEP.

Using the Keyboard

Whether you're a lightning-fast touch typist or the hunt-and-peck variety, you'll find an old, familiar friend at the center of the keyboard. It's the same layout (called QWERTY, for the arrangement of these keys) that's been on nearly every typewriter, Teletype, and computer manufactured in the last 100 years.



Look at the rest of the keyboard. Do you see a lot of keys that aren't on a typewriter? These are special keys to help you enter and control your programs. We'll tell you about a few that you need to know now:

The [ENTER↵] Key

Over on the right, where your little finger can find it easily, is the [ENTER] key. It's usually marked with a hooked arrow: ↵.

Depending on your computer, this key may be labeled as [CARRIAGE RETURN], [END LINE], [RETURN], or [CR]. These are all different names for the same key. No matter what computer it's on, it often has the hooked arrow ↵, too.

As you work through this course, if you don't have an [ENTER↵] key, just press the appropriate one on your keyboard every time you're asked to press [ENTER↵] or ↵.

The [ENTER,↵] key is the most important key on your computer. It's how you tell the computer you're finished *entering* a typed line or a command to do something.

To see how [ENTER,↵] works:

1. First type these characters:

CAT

In HP BASIC, this tells the computer to show you a list (a "catalog") of all the programs that are now in the computer's memory. But nothing happens yet.

2. Now press [ENTER,↵]. The screen display changes to something like this:

```
:CS80, 1500, 2
VOLUME LABEL: HPW_C
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
REVID ASCII 2 256 16
SYSTEM_BAS SYSTM 2363 256 18 7-Sep-87 13:15
COUNT_PRT ASCII 1 256 2390 3-Mar-88 9:55
COUNT_DISP PR0G 2 256 2392 3-Mar-88 9:47
```

EDIT	Continue	RUN	SCRATCH	LOAD	LOAD BIN	LIST BIN	RE-STORE
------	----------	-----	---------	------	----------	----------	----------

When you pressed [ENTER,↵], you told the computer you were finished typing the line. The computer then did as you ordered – it produced a "catalog" of programs, complete with information about each program.

[SHIFT] and [CAPS LOCK]

The [SHIFT] key lets you type capital letters or the symbols that are on the upper part of a key. For example, to type an asterisk:

1. Press and hold the [SHIFT] key.
2. Press the number 8 key on the upper part of the keyboard. The computer displays:

*

To "lock" the keyboard so it types only capital letters, press the [CAPS LOCK] (it may be called CAPS) key. Press the key again to "unlock" capital letters. On most keyboards, a [CAPS LOCK] lamp lights to show when the keyboard is locked for capitals.

Unlike a typewriter's lock, the [CAPS LOCK] key locks *only* the 26 letter keys. It doesn't affect the number keys or other keys.

Number Keypad

Way over on the right side of most keyboards you'll find a "10-key pad." That's because it has 10 number keys (0 through 9) and arithmetic keys in a special arrangement that's easier to use for many people. (It's the same key arrangement you'll find on an adding machine or calculator.)

Some keyboards have a [NUM LOCK] key. This functions like [CAPS LOCK], except it "locks" the number keypad for numbers; or "unlocks" it to allow other functions on these keys (such as [HOME], [PG DOWN], and the arrows). If you have trouble using either numbers or keypad functions, be sure to check the setting of [NUM LOCK] – it may be "locked" when you don't want it to be.

[CTRL]

This is the control key. It may be written as [CONTROL] on your keyboard. You use it in combination with other keys. (If you have a Vectra or IBM-compatible personal computer with the HP BASIC Language Processor card installed, you use [CTRL] to perform some BASIC functions you see on the keyboard overlay, but don't see on the keyboard.)

[Alt]

If you have an HP Vectra PC or other IBM-compatible computer, you'll need the ALTERNATE or [Alt] key. This is another special key, used with other keys to perform BASIC functions. You'll find the keys used with [Alt] on the keyboard overlay.

Softkeys

Along the top of the keyboard, or the left side (or both), you'll see several keys numbered f1, f2, f3, or perhaps k1, k2, k3, and so on. These are the user-definable, or "soft" keys.

The functions of these softkeys are defined by the user—that is, by you. When you first load the HP BASIC language program in your computer, the softkeys are automatically assigned certain functions by that program. For example, f7 may perform the function LIST BIN.

As you'll see, you can easily change the assignments of these softkeys. You can assign any function or group of keystrokes to any softkey.

Every softkey actually has two functions, just as do other keys:

- If you press the softkey and nothing else, you execute one function.
- If you press SHIFT and the softkey, you execute another function.

You'll understand this better when you see how the softkey functions appear on the display.

Whither QWERTY?

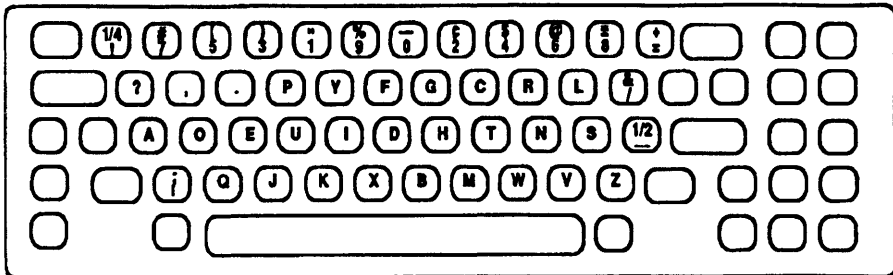
If you think the QWERTY arrangement of keys on your keyboard isn't the most efficient for typing, you're right. How did this layout, which requires both hands and several fingers to type even simple (and very common) words such as "and" and "the" come to be adopted?

The story begins in 1873. Christopher Latham Sholes and Carlos Glidden are turning out the first production models of a new office machine they call the "Type-writer." On the keyboard, keys are arranged in alphabetical order; since most people already know their alphabet, it is thought this will make learning the machine that much easier.

Problems soon develop, however. As typists gain speed, Sholes sees that many often-used combinations of letters cause keys to clash and jam the machine. He commissions a complete redesign, so that on most combinations, keys come up from opposite sides to strike the paper in the center.

The result is called the QWERTY arrangement, named for the keys at the upper left of keyboard. Although Sholes promotes it as being more efficient, the truth is that from its very beginning, QWERTY was actually designed to make typing slower.

After a widely publicized type-off between a touch typist using the Sholes keyboard and a "hunt-and-peck" typist with a competing design, the Sholes layout gains wide acceptance. In 1905, the QWERTY arrangement was adopted as the industry standard – largely due to the efforts of teachers of typing, who had the greatest interest in maintaining the status quo.



The Dvorak Keyboard

Over the years, a number of other, more efficient layouts have been proposed. Dr. August Dvorak's keyboard, to take one example, has been shown to give 30 percent more speed with fewer errors and less fatigue.

Although QWERTY is still solidly entrenched, it makes less sense in the age of computers. Since it's relatively easy to "rearrange" the keyboard on a computer, perhaps you'll want to experiment with other keyboard layouts.

The Display

Look at the computer's display screen now. Seems complex, doesn't it? But the truth is, there are just a few parts to this screen. They are:

- Output area.
- Display line.
- Keyboard area.
- Message and results line.
- Softkey labels.

Output Area →

Display Line →

Keyboard Area →

Message and Results Line →

Softkey Labels →

:CS80, 1500, Z							
VOLUME LABEL: HPW_C							
FILE NAME	PRO TYPE	REC/FILE	BYTE/REC	ADDRESS	DATE	TIME	
REVID	ASCII	2	256	16			
SYSTEM_BAS	SYSTM	2363	256	18	7-Sep-87	13:15	
COUNT_PRNT	ASCII	1	256	2390	3-Mar-88	9:55	
COUNT_DISP	PROG	2	256	2392	3-Mar-88	9:47	
—							
EDIT	Continue	RUN	SCRATCH	LOAD ***	LOAD BIN	LIST BIN	RE-STORE

Output Area

This is the largest part of the display. It takes up most of the screen. And there's more—although you see only 18 lines, there are actually 39 lines in the output area. The screen is like a "window" on the output area.

To see how the window works, try this:

1. Type the following:

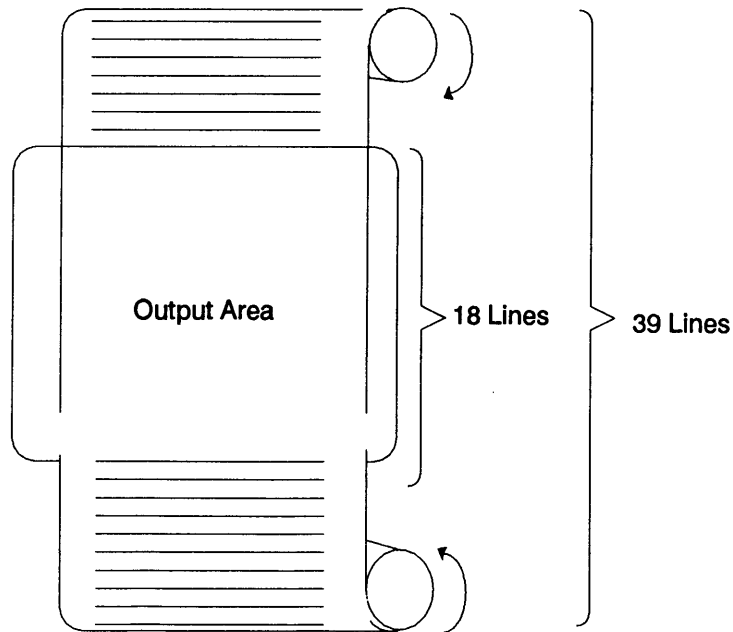
CAT_↓

CAT_↓

CAT_↓

This fills the output area with information – a repeated catalog of all HP BASIC programs now in memory. (If you haven't written any programs yet, you'll still see a few lines containing system information.)

2. Press [↓], the down arrow key. See the list of programs move down through the display?
3. Now press and hold [↑], the up arrow key. See the catalog listing move the other way? It may move right out of sight.



At any time the screen can show only part of the output area. But use the up or down arrow key, and presto! Everything in the output area "scrolls" through the window.

When you were scrolling the window, did you notice that the bottom part of the screen didn't change? This bottom area is reserved for other uses: the display line, keyboard area, message line, and softkey labels.

Display Line

This line is used for prompts from a program to you, the operator. When a program needs some information from you (your name, for example), it asks you for the information here. You'll see this as you write programs that prompt for input values.

Keyboard Area

See the underscore mark or small highlighted area on the screen? This is called the "cursor." It's a place mark that shows where the next character you type will appear. On some computers, the cursor blinks.

Right now the cursor is at the beginning of the keyboard area. This area is actually two lines. To prove it:

1. Put your finger on the "E" key and hold it down. Watch the sturdy little E's march across the screen.

```
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE_
```

2. Keep holding the E key down. When they reach the screen's edge, the E's, undeterred by this artificial boundary, wrap around to the next line and keep right on marching.

```
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE  
EEEEEEEEEEEEEEEE_
```

This shows something about HP BASIC: You can type statements that are up to dozens of characters long.

Message and Results Line

Now that you have the keyboard area filled with E's, you're ready to look at the message/results line. Press [ENTER,↵]. The screen displays an error message:

```
ERROR 935 Identifier is too long
```

When you pressed [ENTER,↵], the computer thought you wanted to "enter" an HP BASIC command. But all those E's weren't anything the computer could recognize. So it did nothing, and told you you'd have to change that line to something it could recognize.

Error messages and the results of keyboard operations always appear in the message/results line.

Incidentally, do not fret over error messages. They don't hurt the computer or your programs. They're just the computer's way of telling you to fix a mistake before going on.

Softkey Labels

The bottom line on the screen shows softkey labels for these user-definable keys. The left label is for key f1, the next for key f2, and so on.

Here's an example of softkey labels:

```
EDIT  Continue  RUN  SCRATCH  LOAD ""  LOAD BIN  LIST BIN  RE-STORE
```

On this keyboard now, the softkeys are "typing aids" for HP BASIC commands. To type the command LIST BIN (listing all the binary programs available to HP BASIC), you'd press key f7.

The "default" softkey labels and functions (the ones you see when you first turn the computer on) are different, depending on your computer and the version of software you have. HP 9000 Series 300 computers, for example, give you a display like the one above; you can summon different displays with the [Menu] and [User/System] keys on the keyboard. If you have an HP 9000 Series 200 computer, you'll see 10 softkey labels corresponding to softkeys numbered k0 through k9. Your computer's documentation is the place to look for more information.

Remember, you can change the labels and the functions of the softkeys. Later on in this course you'll learn how to do just that.

Erasing What You Don't Want

To erase a character you don't want, press the [DEL CHR] (delete character) key. This may also be called [DEL C] or [Delete char] or simply [DEL]. Try it now on an E:

1. Use [BACK SPACE] or the left arrow key [←] to move the cursor under one of the E's in the keyboard area.

```
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEE
```

2. Press [DEL CHR]. One E is erased.

```
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEE_
```

3. Now use [BACK SPACE] or the left arrow key to move the cursor into the middle of the line of E's. (Anywhere in the line will do.)

```
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEE
```

Hold down the [DEL CHR] key and watch the cursor gobble up E's. Like most keys, the [DEL CHR] key automatically repeats if you hold it down.

CLEAR LINE

The [CLEAR LINE] key or a CLEAR LINE statement erases a line at a time. To see how it works:

1. Make sure the keyboard area on the screen is filled with E's.
2. Press the [CLEAR LINE] key.

The entire keyboard area is cleared; that's because the computer sees this entire area as a single line.

CLEAR SCREEN

You can clear the entire screen with the CLEAR SCREEN statement or the [CLR SCR] key. Try it now.

When you execute the CLEAR SCREEN command, everything disappears from the screen except the cursor. This affects only the screen — not the computer's memory, or any programs or anything else.

If you're uncomfortable staring at the trackless void of a blank screen, type CAT, ↵ to see a catalog of programs again.

KEY LABELS ON/OFF

If you decide you don't want the softkey labels at the bottom of the screen, use the KEY LABELS OFF command to turn them off. Use KEY LABELS ON to turn them on again.

This doesn't affect the softkeys themselves; it merely turns the labels on or off on the screen. Try this:

1. Make sure the key labels are on the screen, then press one softkey. When you press the softkey, its function is "written" in the keyboard area, just as if you had typed it there. Try this one, if it's on your softkey display:

LIST BIN

2. Press [ENTER_↵]. You can see that the function is executed.
3. Now turn off the softkey labels: Type KEY LABELS OFF_↵. The key labels are gone.
4. Press the same softkey again. The results are the same. Whether the key's label is displayed or not, the key function operates normally.
5. To see the softkey labels again, type:

KEY LABELS ON_↵

Math Functions

Just like a pocket calculator or adding machine, HP BASIC gives you a number of mathematical functions. Later you'll see how to use these in a program to calculate answers automatically. For now, though, learn to use these math functions from the keyboard.

Keeping a Trail of Your Work

To keep a record of your calculations on the screen, type this:

PRINTALL IS 1_↵

Then press the [PRT ALL] key.

This keeps a record of your keyboard operations on the screen, so if you make a mistake, you can see what you did wrong.

Addition

First, something simple: to add 2 and 2, just type 2 + 2 [ENTER,↵]. HP BASIC does the rest. Try it!

1. Type:

2 + 2

2. Press [ENTER,↵]

The answer appears on the message/results line:

4

If your keyboard has more than one + sign, it doesn't matter which one you use. You can press either the + sign on the upper right-hand portion of the keyboard or the + sign on the numeric keypad if your computer has one.

Subtraction

To subtract one number from another, use the hyphen (–) or the minus sign on the numeric keypad. For instance, to subtract 13,771 from 18,405:

1. Type:

18405-13771

2. Press [ENTER,↵].

The computer displays the result:

4634

Multiplication

To multiply two numbers together, you don't use a multiplication sign (x); instead, use the asterisk (*). For example, to multiply 12 x 185, type:

12 * 185_↵

The computer displays the answer:

2220

Division

In place of the division sign (÷), use the slash (/) to divide one number by another.

Note



Don't try to divide two numbers with the back slash (\). This character has other uses in your computer; it doesn't perform division.

The slash you use to divide two numbers is in the lower right corner of the keyboard, with the question mark over it. It may also be on the numeric keypad.

For example, try dividing 78,234 by 1,250, type:

78234 / 1250_↵

The computer displays the result immediately:

62.5872

Raising a Number to a Power

To raise a number to a power, use the ^ sign. For example, to calculate 2^5 (that is, 2 to the 5th power; or $2 \times 2 \times 2 \times 2 \times 2$), type:

```
2^5_
```

The computer displays the result:

```
32
```

Exponents and Powers of 10

To see an example of an exponent display, calculate 2^{20} by typing:

```
2^20_
```

The result is displayed as:

```
1.048576E+6
```

The "E" (for "exponent") shows that the next number after it is an exponent – that is, a power of 10. So this number is really 1.048576×10^6 . (That is, 1.048576×1000000 , or 1,048,576.)

HP BASIC also shows numbers with negative exponents. Always use parentheses around negative numbers and exponents. To see why, enter this simple-looking expression: -1^2 .

Type:

```
1 ^ 2 _
```

The computer displays:

```
1
```

But is -1 the correct answer? No! The right answer is 1 . (Multiplying two negative numbers together always gives a positive number.)

What happened? The computer sees -1^2 as $-(1^2)$. You should always use parentheses so the computer can't make a mistake. To do it right, type:

```
(-1)^2,↵
```

Now the computer displays the correct answer:

1

Use parentheses for negative exponents, too. For example, to calculate 10^{-12} , type:

```
10 ^ (-12),↵
```

The computer displays:

1.E-12

This display means " 1.0×10^{-12} ".

Other Math Functions

There are a number of other mathematical functions you can use in your programs. These produce an answer based on what is in parentheses after the function.

Take the square root function, for instance: when you type `SQRT (77)`, the computer calculates the square root of `77`:

```
SQRT (77),↵
```

The answer is displayed:

8.77496438739

In this example, the number 77 is called the *argument* of the function. Most functions need an argument.

If you don't want all those trailing decimal places in your answer, use another function: PROUND. Try it with this example again. Type:

PROUND (SQRT(77), -2) ↵

PROUND rounds off the answer to two decimal places:

8.77

Here is a list of some of the general math and trigonometric functions in HP BASIC. You can use all of them from the keyboard. Or you can use them in your programs.

Function	What It Does
ABS	Calculates the absolute value of an argument.
ACS	Returns the arccosine of an argument.
ASN	Returns the arcsine of an argument.
ATN	Returns the arctangent of an argument.
COS	Returns the cosine of an angle.
DEG	Sets the degrees mode.

Function	What It Does
DIV	Divides one argument by another and returns the integer portion of the quotient.
DROUND	Returns an expression's value, rounded to a specified number of digits.
EXP	Raises the base e (2.71828182846) to a specified power.
FRACT	Returns the fractional portion of an expression.
INT	Returns the integer portion of an expression.
LGT	Returns the logarithm (base 10) of an argument.
LOG	Returns the natural logarithm (base e) of an argument.
MAX	Returns the largest value in a list of arguments.
MAXREAL	Returns the largest number available.
MIN	Returns the smallest value in a list of arguments.
MINREAL	Returns the smallest number available.
MOD	Returns the remainder of integer division.
PI	Returns an approximation of π .
PROUND	Returns the number rounded to the specified power of 10.
RAD	Sets the radians mode.

Function	What It Does
RND	Returns a pseudorandom number from a seed.
RANDOMIZE	Randomizes the seed used by the RND function.
SGN	Returns the sign of an argument.
SIN	Returns the sine of an angle.
SQR or SQRT	Returns the square root of an argument.
TAN	Returns the tangent of an angle.

What About Longer Expressions?

It's very nice to be able to add 2 and 2. But the real usefulness of any computer lies in its ability to quickly crunch through long, complex expressions. Let's see how HP BASIC handles longer expressions.

For one thing, you must always use a function or an arithmetic sign before a set of parentheses. Thus, HP BASIC does not recognize this:

$2(3+4)$

But if you add an asterisk before the parentheses, you tell the computer to multiply the quantity inside the parentheses by 2. Try it:

$2*(3+4)$ ↵

The computer displays:

14

Another thing to remember is that HP BASIC performs operations according to a definite order. Here it is:

Step Number	Operation
1	Parentheses: The first thing HP BASIC does is calculate everything in parentheses, working left to right.
2	Functions: Next, functions are calculated.
3	Exponentiation: Next, all exponents (^) are calculated.
4	Multiplication (*) and division (/) are performed next.
5	Addition (+) and subtraction (-) are performed after everything else.

You can see the difference with this simple test:

If You Type:	The Answer Is:
$2*3+4$	10
$2*(3+4)$	14

See the difference? When you typed $2*3+4$, the computer performed multiplication first ($2*3=6$) and did addition last ($6+4=10$).

However, when you added the parentheses, the computer first calculated what was inside the parentheses ($3+4=7$), then multiplied ($2*7=14$).

What about this one: $4[28 - 3(5 + 3)]/2$?

Brackets such as [and] aren't used for mathematics in HP BASIC (they have other duties to perform). But you *can* place one set of parentheses inside another one, like this:

$$4 * (28 - 3 * (5 + 3)) / 2$$

This is called "nesting" parentheses. Like a good scout, HP BASIC follows its rules to calculate this expression, so the answer is:

8

There is virtually no limit to how deep you may nest parentheses (that is, how many layers of parentheses you may have). There's just one cardinal rule:

Cardinal Rule 

You must have an equal number of left and right parentheses.

Fun Functions

Want to find out the date or the time? Want to play music on your computer? HP BASIC has a way to do all of this.

For instance, TIMEDATE returns the current value of the real-time clock in the computer. Use it together with TIME\$ and DATE\$, like this:

TIME\$(TIMEDATE)

This gives the current time:

10:01:31

For the date, type:

```
DATE$(TIMEDATE)↵
```

You see today's date displayed on the screen:

2 July 1988

What about sound? Use the BEEP function; type:

```
BEEP 200, 1↵
```

You hear a looong, looww tone. The first number (200) is the frequency. The number after the comma (1) is the duration in seconds.

Now try this one:

```
BEEP 1000, .5↵
```

This time the tone is higher (1000 Hertz) and shorter (1/2 second).

Review Quiz

Take this quiz to see how well you've learned the material in lesson 1. Try to do the test without referring to the earlier explanations.

When you've finished the quiz, check your answers against those in appendix A. Then go back and review any areas where you had difficulty.

1. You're helping a new user learn HP BASIC. He wants a catalog of the programs in the computer, so he types the following characters:

CAT

Nothing happens. What did he forget to press?

2. When you turn on your computer, the bottom of the screen looks like this:

```
EDIT  Continue  RUN  SCRATCH  LOAD ""  LOAD BIN  LIST BIN  RE-STORE
```

What key would you press for the command RE-STORE?

3. Dozing, you accidentally set your elbow on the keyboard and fill the keyboard line with Z's. How can you erase all those Z's easily, *without* the DEL key?
4. Conscientious city planner Madeline Bray is comparing the population densities of New York and Tokyo. She finds New York's density by dividing the number of people (14,598,000) by the area (1,274 square miles):

```
INT(14598000/1274) ↵
```

```
11458
```

However, when she tries to find the population density of Tokyo, by dividing the number of people by the area, she has a problem. Here is what she types:

`INT(25434000\1089)↵`

Can you help her? What has she done wrong?

5. How would you solve this equation?

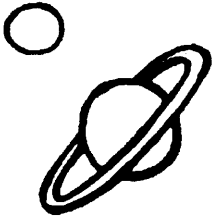
$$10 \left(5 - \frac{3^4}{15 - 3(37 - 14)} \right)$$

Your First Program

This lesson shows how to write and edit a simple HP BASIC program. You will learn about:

- SCRATCH.
- Commands and statements.
- EDIT mode.
- Running a program with RUN.
- Running a program with STEP.
- INPUT.
- Variables.
- RENumber.
- COPYLINES and MOVELINES.
- DISPlay.

Solving a Problem in Calculator Mode



If you worked through lesson 1, you know how to use math functions on the computer keyboard. Everything in lesson 1 was done in BASIC's calculator mode. This means that whatever you press on the keyboard is executed immediately.

Let's solve a problem in calculator mode, just to refresh your memory.

Example: The nine planets in our solar system aren't perfectly round. Still, if you know a planet's diameter, you can get an approximation of the area of its surface by using the formula for the surface area of a sphere:

$$A = \pi d^2$$

Where:

- A is the surface area of a sphere.
- π is the quantity pi (3.14159...).
- d is the known diameter of the sphere.

The earth, for instance, has a diameter of 7926.41 miles. If we assume it is perfectly spherical, what is its surface area?

Solution: At the HP BASIC keyboard, you can solve this problem in calculator mode with a few keystrokes. Just type:

```
PI*(7926.41^2),_
```

The answer is:

1.97379906233E+8

This means 1.97×10^8 ; or 197,379,906.233 square miles.

Writing a Program

If you know the diameters of all nine planets, you can calculate their surface areas using the same technique. But why press all those keys over and over when you have a computer to do the work? Let's write a program that will calculate the surface area of *any* sphere.

What Is a Program?

A program, whether in BASIC or any other computer language, is just a list of statements that are executed by the computer. They're executed in order, one by one, just as if you were entering them from the keyboard. But the computer is automatic and much, much faster.

Edit Mode

Up to now, you've used BASIC only in calculator mode – whatever you typed in was executed immediately, just as if you were using a desktop calculator. Now you're going to switch to edit mode to enter a program.

For this example, first make sure there are no other programs ready to run in the computer. Type:

```
SCRATCH_↵
```

SCRATCH (you may have to type SCRATCH A) erases any other program that may be in the computer. It "clears memory" to a blank slate for you to put in your own program.

The first thing you must do to write a program is to "switch" the computer to edit mode. To do this, press the [EDIT] key. Or simply type the command:

```
EDIT_↵
```

The computer displays the line number of the first line in your program, inserts blank spaces, and places the cursor ready to type the line:

```
10 _
```

Line Numbers

The computer keeps track of your program's statements by means of *line numbers*. Each statement has a line number; when the computer executes the program later, it executes line number 1 first, then line number 2, then 3, and continues until it reaches the end of the program.

Unless you tell it a different way to number lines, the computer automatically gives the first statement a line number of 10, the second statement 20, and so on. This makes it easier to go back later and insert extra lines in your program.

Now type the first line of the program:

```
INPUT D_
```

When you press [ENTER_] in edit mode, the current line is checked to make sure it's something the computer can understand. Then it's entered in the program.

The computer understands you've typed a program line if the line has a line number, then at least one space, then a *keyword*. A keyword is something—like INPUT—that has meaning to the HP BASIC system.

(Incidentally, you can type keywords as either upper- or lower-case letters; the computer automatically changes them to upper case for you.)

After the computer enters the line, it automatically types the next line number for you, and moves the cursor so you're ready to type again.

```
10 INPUT D
20 _
```

If you don't press [ENTER_], the line is forgotten. Here's another cardinal rule:

Cardinal Rule 

If you type or change a line, it isn't entered in the program until you press [ENTER_].

Finish typing the program, just as it's shown here. If you make a mistake, use [BACK SPACE] and/or [DEL CHR] (or the equivalent on your keyboard) to "back up" and correct it. Remember to press [ENTER_] after each line.

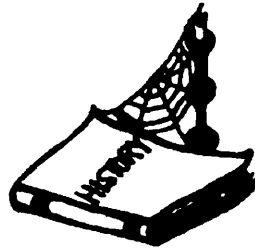
```
10 INPUT D
20 LET A=PI*(D^2)
30 DISP A
40 END
50 _
```

Your program to calculate the area of a sphere is now in the computer. Now all you have to do is *run* the program.

A Bit of BASIC History

Unlike spoken languages such as English and Chinese, BASIC *does* have a known birthdate: May 1, 1964. That's when two faculty members and a handful of students at Dartmouth College first saw correct answers emerge using a new language they had created for their timesharing computer system.

Before BASIC, most programming used batch processing: a student or engineer would write a program, punch out a card for each instruction, then hand over the pack of cards to a computer center. Then, hours (or days) later, the cards were returned, along with a printout showing error codes. It was tedious and time-consuming, usually requiring several "runs" to get the program right.



Those pioneers at Dartmouth created an alternative. They called it BASIC, for Beginner's All-purpose Symbolic Instruction Code. From the outset, it was clear that the word "Beginner's" was a misnomer, since BASIC quickly found favor with a full spectrum of the scientific and technical community.

For one thing, it was easier to use: statements were easy-to-remember words like GO TO, PRINT, and IF...THEN. There was one instruction per program line, and each line began with a keyword. Another advantage of BASIC was its instant feedback: when a user on the timesharing system typed a program line, it was checked immediately for syntax. Error messages popped up right away if the statement was wrong, so correction could be immediate.

Finally, BASIC was hardware-independent. A user didn't have to know anything about file structure to save a program, for instance.

Those early BASIC programmers had a tiny vocabulary to work with—just over a dozen commands and statements in all. By contrast, today's HP BASIC boasts more than 300 keywords!

Running a Program

When you run a program, the computer begins with the first line of the program and executes the statement in that line. Then it moves to the next line and executes that statement, then the next – just as if you were executing them from the keyboard. This is known as *run mode*.

RUN The easiest way to run a program is with the [RUN] key.

Here's how to run the program to calculate the area of a sphere:

1. Press [RUN]. The computer displays:

?

The question mark means the program is halted, awaiting input; in this case, it's waiting for you to enter a value for D, the diameter.

2. Type a diameter, then press [ENTER_↵]. For instance, to calculate the surface area of a sphere the size of Earth, type:

7926.41_↵

The computer finishes running the program and displays the answer:

1.97379906233E+8

You can see how much faster the HP BASIC program could calculate the area of a sphere. Try calculating the surface areas of these other planets:

Planet	Diameter in Miles
Mercury	3100
Mars	4200
Jupiter	88,000

To calculate the surface areas, just run the program and input the diameter for each planet; the computer does the calculations for you:

RUN_

?

3100_

3.0190705401E+7

RUN_

?

4200_

5.54176944093E+7

RUN_

?

88000_↵

2.43284935094E+10

Although you couldn't see it because the program runs so fast, BASIC displays a small square in the lower right portion of the screen when a program is running. This square is the run indicator (or run light). You'll learn more about it in the next lesson.

If the Program Doesn't Work

If the program doesn't seem to run correctly, don't worry—even if you made a mistake entering the program, you can't hurt the computer or other programs in memory. Should you encounter problems, double-check the program listing to make sure you entered everything correctly.

Then try the trouble-killers. (They're listed in the introduction to this course.)

Finally, erase the current program and reenter it, by typing:

SCRATCH_↵

CLEAR SCREEN_↵

EDIT_↵

Enter the program again, exactly as it's supposed to be written.

Parts of the Program

Now let's take a closer look at what happens when you run the program. Just to make things easier to read, clear the screen, either with the [CLEAR SCREEN] key or by typing:

CLEAR SCREEN_↵

To see every step of the program as it runs, you'll use the [STEP] key.

Statements and Functions vs. Commands

In BASIC, anything that can be part of a program – that is, given a line number – is called a statement. You can execute most statements from the keyboard, of course, but they're usually executed as part of a program. For example, each line in your program contains a statement: INPUT, LET, DISP, and END are all statements.

A function, such as PI or SQRT (square root), can be used as part of a statement. And like statements, functions can be executed directly from the keyboard or as part of a program.

A command is different. A command is an order from you that tells the computer to change status, execute a program, etc. For example, RUN, EDIT, and DEL are all commands. In general, you cannot store a command as a line in a program.

Here is how you use functions, statements, and commands:

Question	Functions and Statements	Commands
Part of program?	Yes	No
Use line numbers?	Yes	No
When executed?	When program is run, or from keyboard	From keyboard

Using the [STEP] Key

The [STEP] key is like [RUN], except it moves slowly – one "step" at a time. [STEP] lets you see exactly how a program works. (It's great for finding "bugs" in your programs, too!)

INPUT

Press [STEP] now to see the first line of the program:

```
10 INPUT D
```

Line 10 is an input statement. It means "pause for input from the keyboard." When a running program executes line 10, it pauses, signals you with a question mark, and waits for you to type the value for the diameter, D.

Press [STEP] again to execute line 10. The screen displays:

```
?
```

You can see line 10 was executed and the computer is now waiting for you to input a diameter. Let's use the diameter of Mercury:

```
3100.┘
```

You just typed in the value of a *variable*, D.

Variables

At the beginning of this lesson you used the keyboard to solve for the surface area of a sphere. In calculator mode, you simply typed numbers – such as the diameter of a sphere – whenever they were needed.

A running program, however, keeps track of numbers by means of variables, such as D. By changing the value of the variable, you can change the outcome of the program.

You can name a variable almost anything you want. It's easiest if you use names that relate to what the variable does. For example, in our example program, we used D for diameter, A for area.

You'll learn more – much, much more – about numeric variables in lesson 4. But for now, just remember:

- Use variables for all numeric quantities in your programs.
- Give variables only meaningful names.

LET

You can now see the second line of the program:

```
20 LET A=PI*(D^2)
```

The LET statement assigns a value to a variable. In line 20, the variable for area, A, is assigned the value $\pi \times D^2$.

The LET statement always has an equals sign to show what value you're assigning to the variable.

Incidentally, the actual keyword LET is optional. So these two statements are exactly the same:

```
10 LET A=PI*(D^2)
10 A=PI*(D^2)
```

Here are some examples of LET statements:

```
30 LET X=100
```

Line 30 assigns the value 100 to the variable X.

```
70 Sum=X+5
```

Line 70 adds 5 to the current value of X and places the result in the variable called Sum.

```
10 A=PI*(D^2)
```

Here, line 10 assigns the value πD^2 to the variable A.

```
100 LET I=I+1
```

In line 100, the LET statement increments (adds 1 to) the variable I.

As you see, a variable name is like a person's name—it begins with a capital letter.

The DISP Statement

Press [STEP] again to execute line 20 and see the third line of the program:

```
30 DISP A
```

This means "display the current value of the variable A."

The display is always on the display line of the computer screen. You can use the DISP statement to display variables and other information.

If you enclose something for display in quotation marks, DISP shows exactly what's written. For example, look at this program segment:

```
30 Dickens=100
40 DISP Dickens
50 DISP "Dickens"
```

When the program segment is run, line 30 stores the value 100 in a variable called Dickens. Line 40 displays the value of the variable:

```
100
```

Line 50, though, gives this display:

```
Dickens
```

Later on in this lesson you'll use the DISP statement to make your program friendlier and easier to read.

The All-Important END

In any program, there is just one statement that's required – that the program cannot exist without. It's the END statement.

Press [STEP] again to see the last line of your program:

```
40 END
```

END tells the computer that it's the last line of the program. When the computer executes line 40, it halts and switches back to calculator mode, ready to run the program again.

This brings us to another cardinal rule:

Cardinal Rule 

Every program must have an END statement as the last line.

Press [STEP] one last time to execute the END statement and end the program.

Editing Your Program

Now that you've run your program a few times, you've seen that it easily handles the job of calculating the area of a sphere. But the program is not at all "friendly" or easy to use.

Think about it: If you came back to this program and ran it after several months, would you remember what the question mark meant? Would you have to read through every step to understand what the program does?

Let's alter your program to make it better – and give you some editing practice to boot!

Editing a Line

The EDIT command puts you back in edit mode, ready to type new program lines or edit old ones. Use EDIT now to go right to the first line you want to edit; type:

```
EDIT 30_
```

You see the computer switched to edit mode – and you're right on line 30!

```
10 INPUT D
20 LET A=PI*(D^2)
30 DISP A_
40 END
```

Now change line 30 to read:

```
30 DISP "The surface area is";A
```

You have a number of ways to edit this line:

- You can simply type over the characters of the old line.
- You can turn INSERT on (press the [INSERT] or [INS] key). This means that new characters don't overwrite old ones; instead they're inserted in the line. (To turn INSERT off and go back to overwrite, press [INSERT] or [INS] again.)
- You can type a new line with the same line number: move the cursor all the way to the left and type the new line number. Leave a space, then type the statement.

No matter how you edit the line, be sure to press [ENTER_] when you're done.

You can use the cursor movement keys to go to any chosen line and edit it.

Entering New Lines

Now you should be at line 40:

```
40 END_
```

To enter new lines, move the cursor to the left of the screen, and type the new line number. Then leave at least one space, and type the statement. (That's right, type right over the END statement!) Be sure to end the line with [ENTER,↵], like this:

```
50 CLEAR SCREEN_↵
```

See what happened? Because you began the line with a different line number, you didn't affect line 40 – even though you typed over it. Your new line 50 appears after line 40 when it's entered:

```
40 END
50 CLEAR SCREEN
```

Now enter the rest of the new lines, so the program has the lines shown here:

```
10 INPUT D
20 LET A=PI*(D^2)
30 DISP "The surface area is";A
40 END
50 CLEAR SCREEN
60 DISP "Enter a diameter"
70 WAIT 3
80 _
```

Don't worry if the lines don't line up on your screen. It makes no difference to the computer.

Moving Program Lines

You've entered the new program lines, but they're in the wrong place; you want to move them to the beginning of the program, not the end.

For this you'll use the `MOVELINES` command. Move the cursor all the way to the left of the screen and type:

```
MOVELINES 50,70 TO 10 ↵
```

This command "picks up" lines 50 through 70, and moves them to line 10. The other program lines are renumbered and move down to make room for the new lines. Your program now looks like this:

```
10 CLEAR SCREEN
11 DISP "Enter a diameter"
12 WAIT 3
13 INPUT D
14 LET A=PI*(D^2)
15 DISP "The surface area is";A
16 END
```

The `MOVELINES` command is handy for moving one line or any number of lines to a new location.

Another command, `COPYLINES`, is similar to `MOVELINES`. `COPYLINES` inserts a copy of the lines you specify in the new location; the lines also remain in their original location in the program.

Renumbering Lines

You could run your program just as it is now. But most programmers in BASIC use line numbers in increments of 10, as you did before. You can use the REN (renumber) command to renumber the program lines.

Move the cursor to the left and type the REN command. Be sure to erase any other characters on the line, then execute the command:

```
REN_
```

Your program is changed to:

```
10 CLEAR SCREEN
20 DISP "Enter a diameter"
30 WAIT 3
40 INPUT D
50 LET A=PI*(D^2)
60 DISP "The surface area is";A
70 END
```

When using the REN command, you can specify the line number to begin with, to end with, and the interval. Look at these examples:

```
REN 1000
```

This command renumbers lines, assigning line number 1000 to the first line of the program.

```
REN 50, 25
```

Here, lines are renumbered beginning with line 50, in intervals of 25; the first line in the program becomes line number 50, the next 75, then 100, 125, and so on.

REN 10, 10 IN 5

This begins with line 5 and changes its number to 10; then renumbers in intervals of 10.

If you don't specify a line number or interval, the computer begins with line 10 and uses intervals of 10.

The REN command can be extremely useful when you modify a program. For instance, to add 15 new program lines between lines 10 and 20, you'd first want to renumber using an interval of, say 20; that is, REN 10,20. Then you could add the new lines with no problem.

Run the Edited Program

To run the edited program, type RUN or press the [RUN] key. You can run the program now to calculate the surface areas of other planets in our solar system:

Planet	Diameter in Miles
Saturn	71,000
Uranus	32,000
Neptune	31,000

To calculate the surface area of Saturn, press (or type) RUN. The computer displays:

Enter a diameter

Then:

?

Enter the diameter of Saturn:

71000_

The computer gives you a more meaningful answer than before:

```
The surface area is 1.58367685667E+10
```

You can run the program again to calculate the surface areas of Uranus and Neptune, if you like.

Now let's see how the changes you made to the program affect it.

Clearing the Screen

You've changed the program, and the first line of the new program is now:

```
10 CLEAR SCREEN
```

This shows that CLEAR SCREEN is a statement, not a command—you can use it from the keyboard or in a program. If you want to be sure there's no "garbage" on your screen from previous program runs, a CLEAR SCREEN statement early in your program is a good idea.

Waiting, Waiting

The next three lines of the new program show how you can prompt for input without stopping the program. Look at lines 20 through 40:

```
20 DISP "Enter a diameter"  
30 WAIT 3  
40 INPUT D
```

Can you figure out what happens when line 20 is executed? That's right, the characters in quotation marks are displayed exactly as they are written.

Line 30 is a WAIT statement. It causes the computer to wait before proceeding to line 40. Since you've specified WAIT 3, the computer waits three seconds, displaying the words "Enter a diameter." Then execution automatically moves on to line 40 and continues.

Line 40 is our old friend, the INPUT statement. Remember that it displays a question mark and waits for input – and doesn't continue until you type a number and press [ENTER_↵].

A Friendlier Display

After the calculation is performed in line 50, line 60 displays the value for A – the surface area of the sphere. But because you've added the words in quotation marks, line 60 also displays a message.

```
60 DISP "The surface area is";A
```

When line 60 is executed, the computer displays the words inside the quotation marks, exactly as they are written. On the same line, it also displays the current value of variable A. So what you see is:

```
The surface area is 1.58367685667E+10
```

See the semicolon just before the A? This tells the computer to leave just one space before it displays the value of the area. You'll learn more about spacing in lesson 3.

Review Quiz

1. What's the only statement required in a program?
2. How do you put a computer in edit mode for typing or editing program lines?
3. What appears on the screen when this program is run?

```
10 LET Dta=1000
20 DISP "YOU OWE";Dta;"DOLLARS!"
30 END
```

4. What key is used to single-step through a program?
5. Write a short program to compute the squares of numbers. The program should stop to prompt for input, then display both the original number and the answer; as, for example:

```
What is the number?
```

```
The square of 5 is 25.
```


Saving Your Program

In this lesson you'll write and edit a new program. You'll learn to list the program, and you'll learn to direct program output to the screen or the printer. Here's what you'll be studying in this lesson:

- [PAUSE] and [CONTINUE].
- The run light.
- PRINTER IS.
- LIST.
- REMarks and the exclamation point.
- Specifying MASS STORAGE IS.
- Initializing a disk with INITIALIZE.
- STORE, LOAD, and RE-STORE.
- SAVE, GET, and RE-SAVE.
- DISP vs. PRINT on the screen.
- The difference between a semicolon and a comma in the PRINT and DISP statements.

Write a New Program

For this lesson, begin typing in a new program. First, clear any program that might be in your computer, and switch to edit mode. Type:

```
SCRATCH_␣  
EDIT_␣
```

Now type in the following program, exactly as it's written here:

```
10  !This program counts  
20  PRINTER IS 1  
30  FOR I=1 TO 50  
40  DISP "The current value of I is",I  
50  PRINT "The current value of I is";I  
60  WAIT .1  
70  NEXT I  
80  END  
90  _
```

Be sure you use a comma in line 40 and a semicolon in line 50.

Once you've typed in the program, switch out of edit mode and run the program.

Run the Program

To get out of edit mode and run the program, press the [RUN] key. The computer begins displaying the numbers from 1 to 50:

```
The current value of I is 1
The current value of I is 2
The current value of I is 3

The current value of I is 48
The current value of I is 49
The current value of I is 50

The current value of I is      50
```

As you can see, the program counts up to 50, then stops.

[PAUSE] and [CONTINUE]

While the program is running, press [PAUSE]. Execution "freezes" in mid-program, and the display shows the next line to be executed.

Now press [CONTINUE]. The program resumes right from where it left off.


You can use [PAUSE] and [CONTINUE] along with [STEP] to help write and debug your programs.

The Run Light

Do you see the small square in the lower right-hand corner of the screen while the program is running? This is the *run light*.

```
The current value of I is 24
The current value of I is 25
The current value of I is 26
The current value of I is 27
The current value of I is 28
The current value of I is 29
The current value of I is 30
The current value of I is 31
The current value of I is 32
The current value of I is 33
The current value of I is 34
The current value of I is 35
The current value of I is 36
The current value of I is 37
The current value of I is 38
The current value of I is 39
The current value of I is 40
The current value of I is 41

The current value of I is      41
```

Run Light 

The run light tells you the current status of the computer in BASIC. This chart shows what the different displays of the run light mean:

Run Light	What It Means
Blank	Program stopped.
Solid box	Program running.
—	Program paused.
I/O	Program paused, but transferring data.
?	Awaiting input from the keyboard.
*	Executing a command from the keyboard.

Before we store our counting program, let's look at some new statements it introduces.

REMARKS First, examine line 10:

EDIT

```
10 !This program counts
```

Line 10 is a *remark*. When you use an exclamation point or the keyword **REM** as the first item in a line, that line is ignored by an executing program – that is, the line is skipped.

These lines are all examples of remarks, and they're all skipped during execution:

```
10 !This program is useful for calculating the area
20 !of a sphere. When prompted, input the diameter of
30 !the sphere, D. The area, A, is printed.
```

```
100 REM The next section of code produces a graphed output
110 !of the user's biorhythm chart.
```

You use remarks to annotate your programs for others (or yourself). Remarks should tell what the program does and how it does it.

Since they're not executed, you're probably wondering "Why use remarks at all?"

Here's why: As your programs grow in size and number, you won't always remember what each program does. And trying to deduce the meaning of a complex program by a laborious examination of each individual line is an exercise you won't want to do often.

Cardinal Rule

Use remarks throughout every program to document how it works.

Telling Output Where to Go

The next statement, `PRINTER IS`, tells where you want the output from this program to appear. It specifies where the answers or other output will be printed:

```
20 PRINTER IS 1
```

When you specify `PRINTER IS 1`, the number "1" means that any printed output is directed to the computer's display screen. It's actually "printed" there on the screen. You can also use the `PRINTER IS` statement to redirect output to a printer, plotter, or other device.

The `PRINTER IS` statement uses a *select code* to identify which printer or other device will receive what you print. The select code is unique. It is based on which interface (for example, HP-IB or another interface) you're using between the computer and the printer.

(If you have more than one printer attached to the same interface, you'll need a little more information. Besides the interface select code, you'll also need a code for the device itself; this corresponds to numbers set on a switch in the printer or other device.)

Look at these examples of `PRINTER IS` statements:

```
20 PRINTER IS 1
```

Line 20 specifies the computer's internal CRT display.

100 PRINTER IS 26

Line 100 specifies a printer with interface select code 26. This statement can be used with a Vectra or other PC that has a standard MS-DOS printer as the system printer.

PRINTER IS 701

This statement directs printed output to a device with address 01 on HP-IB (interface select code 7).

You can also use some special words to specify some parts of the computer, such as:

440 PRINTER IS CRT

Line 440 redirects output to your computer's CRT display screen. It's the same statement as PRINTER IS 1.

The words you can use are:

Word	Value
PRT	701
KBD	2
CRT	1

Another way to direct printer output is to a variable, such as:

PRINTER IS Laser_jet

You'll learn more about directing output to variables and path names in lesson 10.

The PRINTER IS statement lets you change your output device in mid-program. For instance, you could have a section of code like this:

```
120 PRINTER IS 1
130 PRINT A
140 PRINTER IS 701
150 PRINT A
```

This program segment first prints the value of variable A on the computer's display. Then it prints the same value on a printer with address 01 on HP-IB.

When you first turn on the computer or BASIC, it "wakes up" set to PRINTER IS 1, so line 20 is really unnecessary right now. Later on in this lesson, though, we're going to change line 20 to put our count on a printer instead of the display.

The FOR-NEXT Loop

Lines 30 through 70 make up a "for-next" loop. You'll find more – lots more – about loops in lesson 7 of this course. For now, though, here's a brief explanation of what happens in lines 30-70:

```
30 FOR I=1 TO 50
40 DISP "The current value of I is",I
50 PRINT "The current value of I is";I
60 WAIT .1
70 NEXT I
```

The first time line 30 is executed, variable I is set to 1. Line 40 displays the value of I on the display line. Line 50 then prints the value of I to whatever is the current printer – that is, to whatever is set with the PRINTER IS command.

Line 60 causes the program to pause for 1/10 second. Then line 70 causes execution to go back to line 30 for the "next I," which is 2.

The "loop" repeats, adding 1 to the value of I each time, and printing the value of I when I = 3, I = 4, and so on. When I = 50, execution continues with the END statement in line 80, and the program stops.

PRINT vs. DISP

What's the difference between PRINT and DISP? The DISP statement always puts output on the display line of the computer's screen. PRINT can display output on the screen, or it can send output to a printer, depending on the previous PRINTER IS statement.

PRINT output appears
in output area →

DISP output always
appears here →

```
The current value of I is 24
The current value of I is 25
The current value of I is 26
The current value of I is 27
The current value of I is 28
The current value of I is 29
The current value of I is 30
The current value of I is 31
The current value of I is 32
The current value of I is 33
The current value of I is 34
The current value of I is 35
The current value of I is 36
The current value of I is 37
The current value of I is 38
The current value of I is 39
The current value of I is 40
The current value of I is 41

The current value of I is      41
```

In your mini-program, the PRINT statement (line 50) puts its output in the output area. The DISP statement always produces output in the same place – on the display line.

Comma vs. Semicolon

You probably noticed that the output on the display line is a little different than in the output area. That's because the DISP statement in line 40 has a comma between "The current value of I is" and the variable I, while the PRINT statement has a semicolon in the same location.

```
40  DISP "The current value of I is",I
50  PRINT "The current value of I is";I
```

The comma between elements of a PRINT or DISP statement leaves a wide gap between the elements when they're printed. The semicolon prints or displays them next to each other, with only a single space or a minus sign (for numbers) or no spaces at all (for words). You'll learn a lot more about how to control printed output in lesson 9.

Getting Out of Edit Mode

To get out of edit mode, you can run the program, of course. But if you want to get out of edit mode without running a program, use another key, such as [PAUSE]:

```
[PAUSE]
```

Commands such as LOAD, CAT, LIST, and RUN and keys like [STEP], [PAUSE], and [CLEAR SCREEN] also take you out of edit mode.

Listing Your Program

You have already seen that switching to edit mode displays the program on the screen. To list a program without going into edit mode, use the LIST command. Try it now:

```
LIST_
```

You see a listing of the program currently in memory.

```
10 !This program counts
20 PRINTER IS 1
30 FOR I=1 TO 50
40 DISP "The current value of I is",I
50 PRINT "The current value of I is";I
60 WAIT .1
70 NEXT I
80 END
```

Available memory = 917692

At the bottom of the screen, you can see how much memory (in bytes) you have available in the computer for more program lines.

Partial Listing

To see just a partial listing of a program, just list *from* a certain line number, or from one line number to another. For example:

```
LIST 30, 70 ↵
```

This gives a listing of just lines 30 through 70.

Once a program is listed, use the up and down arrow keys to see the entire listing.

The listing is always sent to the current printer. To make sure your listing appears on the display, type PRINTER IS 1 before you type LIST.

What About LIST BIN?

Another list command available to you is LIST BIN. This is really like CAT; it lists the *binary* programs currently in the computer. These binaries are used for certain tasks by BASIC.

If You Have Printing Problems...

BASIC is supposed to be hardware-independent. This means that, usually, you simply direct your output to a select code (7, for example, or 26, or PRT) with the PRINTER IS statement. Usually, you don't need to worry about how the printer is connected, or what kind it is.

Usually, that's all true.

Computers are unusual creatures, though, and problems do develop. Especially if you are using the HP BASIC Language Processor in a Vectra, IBM PC AT, or other personal computer, you may have to make some changes before you can print.

Here are some trouble-killers for printing:

1. If the program doesn't print, and seems to run forever without stopping, it probably means you've told the program to print to a nonexistent printer. Press the [CLEAR I/O] key to regain control. Then try the other trouble-killers below.
2. Make sure the cable between the printer and computer is plugged in securely at both ends.
3. Make sure the printer's power cable is connected, the printer power switch is on, and the printer has paper.
4. Look at the printer and make sure its ON LINE lamp is on. If it isn't, there may be an [ON LINE] pushbutton; press it until the ON LINE lamp comes on.
5. To find out the printer's select code, type:

```
PRINTER IS 7 ↵  
PRINT "HELLO" ↵
```

If that doesn't work, do the same thing, substituting other select codes after PRINTER IS. Try PRINTER IS 26, PRINTER IS 9; in fact, try *all* of these:

26 701 9 15 19 23 24 25

When the printer finally prints, you've found the select code. From now on, use the correct select code in the PRINTER IS statement in your program.

6. If your printer won't print a short listing or program, you may have to send it a form feed. Type this after your program has run:

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

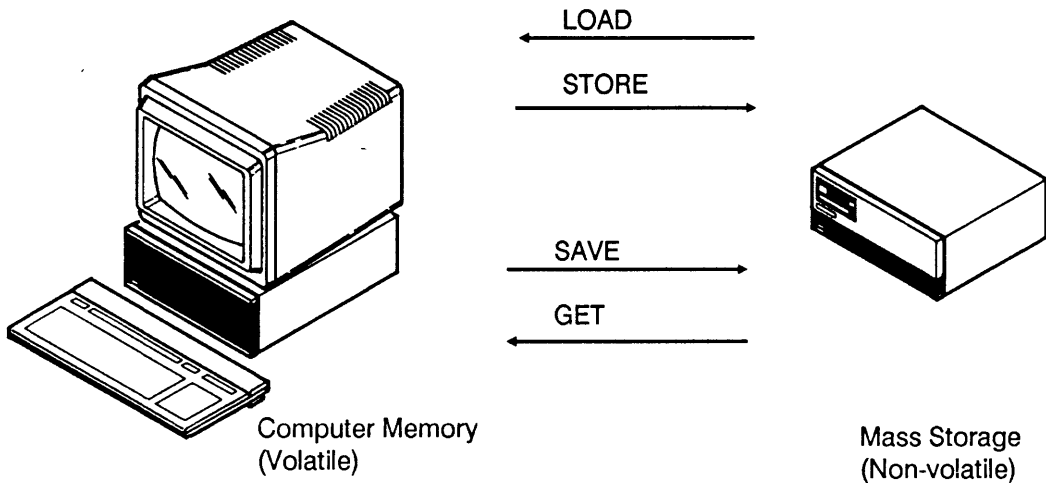
If this produces an output from your printer, include it as a line in your program.

7. If you have a Vectra or other PC and all else fails, you may need to modify the HPW.CON file as described in the documentation for the BASIC Language Processor card. Remember to reboot BASIC after you modify this file. Good luck!

Making Your Program Permanent

Before we continue, you should know a little about your computer's memory. It's measured in *bytes*. The number for "Available memory" you see when you LIST a program shows how many bytes are left for program lines in your computer.

All your computer's memory is *volatile*, which means any program in it is lost when you turn the computer off. Before you turn the computer off or type another program, you should store or save the current program in a mass storage unit.



Mass storage units are usually outside the computer itself. They include disks, diskettes, magnetic tapes, and bubble memory. These mass storage units can contain many programs.

To *run* a program, it must be in your computer's memory; you can type it in or bring it in from a mass storage unit. In general, your computer can have only one program in it at a time.

There are two commands for putting your program onto disk or tape: they are STORE and SAVE, and they differ mainly in the format in which your program is placed into mass storage.

Before you can store or save a program, though, you'll need to tell the computer where to save it, and you *may* need to initialize a disk. Read on!

Specifying Mass Storage Is

A mass storage unit can be a floppy or hard disk, magnetic tape drive, or other unit. It might be located right inside your computer, or attached to it. Or it could be in a remote location and used by many computers.

Your computer probably has at least one mass storage unit, and maybe more. One way to verify the *current* mass storage unit is to do a catalog. Type:

```
CAT_
```

The top line of the CAT display (you may need to scroll down to see it) shows the mass storage unit for which you're seeing the catalog; that is, the default unit.

```
:CS80, 1500, 2  
VOLUME LABEL: HPW_C
```

The way you tell the computer which mass storage unit to use is by "addressing" the unit.

The statement MASS STORAGE IS (you can abbreviate it as MSI) tells your computer what is the current, or default, mass storage unit. Here's an example:

```
MASS STORAGE IS ":CS80,1500,0" ↵
```

This tells the computer that the default mass storage unit – the one that will be used for storing and loading programs – is the drive called CS80, 1500,0. This means drive A (the top or left internal floppy disk drive on most personal computers).

Incidentally, you usually don't have to use the first part of the specifier. So these two statements are the same:

```
MASS STORAGE IS ":CS80,1500,0" ↵  
MASS STORAGE IS ":",1500,0" ↵
```

Notice that you still need the colon and the comma, though.

Here are examples of how you'd address some mass storage units:

This Statement:	Makes This the Default Mass Storage Unit:
MSI ":",1500,0"	Internal drive A in Vectra or other PC
MSI ":",INTERNAL"	Internal disk drive in HP Series 200
MASS STORAGE IS ":",700,0"	External drive at address 0 on interface 7 (usually HP-IB)
MSI ":",1500,2"	Internal hard disk C in Vectra or other PC

Initialize a Disk

If a disk is brand-new, you must *initialize* it before you can use it to store programs or data. Since initializing wipes out *everything* on a disk, any data or files on that disk will be lost. So always do a CAT of the disk before you initialize it.

Cardinal Rule

Always verify that a disk doesn't contain anything valuable before you initialize it.

This procedure is to initialize a floppy disk in drive A. (If your computer is different, use the disk drive you have, and just substitute the correct mass storage unit specifier.)

1. Insert a new floppy disk in drive A. (On most computers, drive A is the top or left-hand floppy disk drive.)

2. Type:

```
MSI "CS80,1500,0" ↵  
CAT ↵
```

You see a catalog of the disk in drive A, so you can decide whether or not you want to initialize the disk. If the disk has *never* been initialized, you'll see a message such as:

```
Medium uninitialized
```

3. Type:

```
INITIALIZE ":",1500,0" ↵
```

The initialization process can take several minutes. Then you'll see a message telling you initialization is completed.

Note



If you are using a Vectra or other IBM AT-compatible personal computer, see the sidebar titled "Initializing a Disk on a PC." You'll find the sidebar later in this lesson.

Here's what INITIALIZE does to a disk:

- Erases all data and checks all bits on the disk.
- Sets up sectors of magnetic media, and establishes interleave pattern between sectors.
- Establishes a volume label and a directory.
- Establishes the amount of memory for storage.

Format and Interleave Factor

When you use INITIALIZE as in this example, without any other specifiers, the computer chooses the best format and interleave factor for you. You can specify interleave factor and format like this:

```
INITIALIZE "Mass storage unit", Interleave factor, Format
```

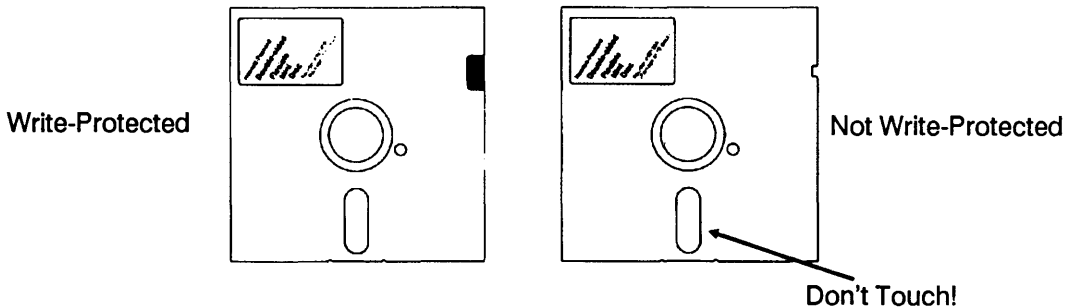
If you're only using the disk in this drive, and if your disks aren't becoming too full, you probably don't need to specify interleave factor and format: the computer automatically makes the best choice for you.

For more flexibility in disk usage, refer to more detailed explanations of disk media and the INITIALIZE command. You'll find these in other manuals for HP BASIC.

About Disks

The disks you're most likely to use are the 5-1/4 inch disk and the smaller but more rugged 3-1/2 inch disk.

BASIC storage capacity of the 5-1/4 inch disk is 360K bytes—in fact, it's over 360,000 bytes of memory. (Just for reference, one typed character, like the letters on this page, occupies one byte.)

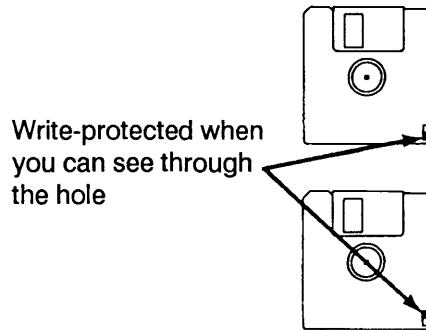


When a tab covers the notch, the disk is *write-protected*. This means you can't copy any files or data to it. You can't initialize it, either. Use write-protect tabs to save your valuable programs and data against being erased or overwritten.

Don't touch the oxide surface of these flexible disks. Just the tiniest bit of dirt or oil from your skin can make it impossible to transfer information.

Although the 3-1/2 inch disk comes in capacities of 720K bytes and 1.44M bytes, HP BASIC can only format 720K bytes of mass storage. (So if you have a 1.44M byte disk, you'll be wasting half the storage capacity of the disk. Save your money!)

The disk has a sliding tab for write-protection. If you slide the tab so you can see through the hole, the disk is protected against writing or initialization.



Initializing a Disk on a PC

If you are using the HP BASIC Language Processor card in a Vectra or other AT-compatible personal computer, you may get this message when you try to INITIALIZE a disk:

FORMAT UTILITY NOT FOUND

This is because on a Vectra or other PC, the DOS FORMAT utility program is called and used by HP BASIC's INITIALIZE statement.

The easy way out: The easiest way of handling the problem is to exit to DOS and initialize the disk in the usual way from DOS: Insert the disk in drive A, type FORMAT A:, and let DOS do it.

Temporary fix: If you want to initialize a disk from HP BASIC, you can add a path to the HP BASIC file right from the keyboard. Do this:

1. Exit to DOS.
2. Type CD\↵
3. Specify a path to the HP BASIC directory. For instance, if HPW is the directory in which HP BASIC is loaded, type:

```
PATH=C:\HPW↵
```

4. To get back to HP BASIC, type :

```
CD\HPW↵  
BASIC (or HPBASIC)↵
```

5. Now use the INITIALIZE statement normally.

You only need to do this temporary fix once each session, although the path is "forgotten" when you turn off power to your computer. So you'll have to perform this procedure every time you turn on your computer.

Permanent fix: If you're going to be initializing many disks, a better way of handling this is to add to the PATH command that's in your personal computer's AUTOEXEC.BAT file.

The AUTOEXEC.BAT file contains a list of paths – places to look for executable files – so the computer knows to look not only in the root directory but also in the directory in which HP BASIC is loaded.

For instance, if HP BASIC is in directory HPW on hard disk C, you'll need to add the following command to the AUTOEXEC.BAT file:

```
PATH=C:\HPW
```

Storing and Loading

The STORE command puts a copy of your program safely away in mass storage, so that if you change the program or pull the plug on the computer, the copy is unaffected.

The program is stored in a *file* in mass storage. Specifically, it's stored in a kind of file known as a PROG file.

To place your program into mass storage, use STORE followed by a file name in quotation marks. For example:

```
STORE "COUNT_DISP" _
```

Your program is now stored as the program "COUNT_DISP".

Note



This stores the program on the current mass storage unit.

Re-Storing the Program

Once a program has been stored, a copy of the program always remains in mass storage. If you want to store a program under that same name again (for instance, if you've modified it), you must use the RE-STORE statement. To re-store the program "COUNT_DISP", type:

```
RE-STORE "COUNT_DISP" ↵
```

(There's also a statement called RESTORE. It is different than the RE-STORE statement.)

What About File Names?

File names can be any combination of letters and numbers, up to 10 characters long. You can't have spaces between characters. You can use uppercase or lowercase letters – and these file names are "case-sensitive." (That is, a file name of "Nautilus" is different than a file name of "NAUTILUS").

Here are some legal file names for programs:

```
DATA_17  
Watch_N_Wt  
Customer_1  
PROGRAM_A
```

Scratching a Program from Memory

Now a copy of your "COUNT_DISP" program is stored permanently in mass storage. It's also still in the computer's BASIC memory. Use the command SCRATCH to erase everything from computer memory:

```
SCRATCH ↵
```

To prove that nothing remains in memory, you can try to LIST the program, or try to RUN it:

```
RUN ↵
```

Since there's nothing in memory for BASIC to run, you get only an error message:

```
ERROR 5   Improper context terminator
```

Don't worry— a copy of your program is safely tucked away in mass storage.

Loading the Program

To load a program from mass storage, use the LOAD command and a file name. Try loading your "COUNT_DISP" program now:

```
LOAD "COUNT_DISP"↵
```

This statement first clears any previous programs from the computer's memory, then brings in the program "COUNT_DISP." You can't see it unless you do a LIST, but the program is loaded. To prove it, just hit [RUN]:

```
The current value of I is 1
The current value of I is 2
The current value of I is 3
```

LOAD can also load and auto-start a program. Try this:

```
SCRATCH↵
LOAD "COUNT_DISP", 10↵
```

Did you see what happened? You know that the computer's memory was cleared because you typed SCRATCH.

But when you typed the next line, LOAD "COUNT_DISP", and followed it with a comma and line number, presto! The program began running.

In fact, a LOAD statement with a line number after it does three things in order:

1. It clears the computer's memory of any previous program.
2. Then it loads the new program.
3. Then it runs the new program beginning with the line number.

And here's another feature of the LOAD statement: it's a statement. Remember what that means? That's right, you can use it *in a program*, like this:

```
110 LOAD "PART2"
```

When line 110 is executed in a program, it clears the computer's memory, loads the program named "PART2", and resumes execution with the lowest-numbered line of the new program. If you specify a line number, execution begins with that line number.

Saving and Getting a Program

Another way to save a program is with the SAVE statement. Like STORE, the SAVE statement places a copy of your program into mass storage. It saves the program in a different format, though. SAVE puts the program into something called an ASCII file.

The ASCII file is different from the PROG file you created earlier with the STORE statement. You'll see this difference later when you look at a catalog.

To retrieve a copy of a saved program from mass storage, you use the GET statement. This is similar to the LOAD statement used with stored programs. To re-save a program in ASCII format after you've modified it, use RE-SAVE (just like RE-STORE).

Seeing a Catalog

If you worked through lesson 1, you already used the CAT command to generate a "catalog" of programs in your current mass storage unit. Let's try CAT again to see how the programs look:

CAT ↵

The computer displays a list of BASIC programs in this mass storage unit. You can see that "COUNT_DISP" is a PROG file. It requires two "records" for the file in memory.

:CS88, 1500, 2		VOLUME LABEL: HPU_C				
FILE NAME	PRO TYPE	REC/FILE	BYTE/REC	ADDRESS	DATE	TIME
REVID	ASCII	2	256	16		
SYSTEM_BAS	SYSTEM	2363	256	18	7-Sep-87	13:15
COUNT_PRNT	ASCII	1	256	2390	3-Mar-88	9:55
COUNT_DISP	PROG	2	256	2392	3-Mar-88	9:47

-

EDIT Continue R/N SCRATCH LOAD "" LOAD BIN LIST BIN RE-STORE

In this example, "COUNT_PRNT" is an ASCII file. It requires only one record for the file. As you probably guessed, PROG files chew up more mass storage space than ASCII files – an important point if you're concerned about saving space on a disk or tape.

Directing Output

The catalog from CAT, and the program listing from LIST, are sent to the current printer—that's the display if you've specified PRINTER IS 1. You can redirect them by changing the printer, or by adding the number sign (#) followed by the printer's select code, like this:

```
LIST #701
```

This prints a program listing on the printer with address 01 on the HP-IB interface.

```
CAT TO #26
```

This prints a catalog on an MS-DOS printer.

Purging a Program from Mass Storage

OK, you're writing and saving (or storing) programs. Mass storage is filling up fast. How do you erase a program from mass storage?

Use the PURGE statement. PURGE erases the named program from the current mass storage unit. (If there's no such program in the current mass storage unit, nothing happens.)

To erase "COUNT_PRNT" from mass storage, type:

```
PURGE "COUNT_PRNT" ↵
```

You can use PURGE for any kind of file—a PROG file, an ASCII file, or a data file. (You'll learn about *those* in lesson 10.)

Review Quiz

1. You want a program to prompt a computer operator—to make an instruction appear in the same place, no matter what else is on the screen. Do you use a PRINT or a DISP statement?

2. When this program is run, what is printed on the current printer?

```
10 PRINT "It was"  
20 !PRINT "the best of times,"  
30 !PRINT "It was"  
40 PRINT "the worst of times."  
50 END
```

3. Which of the outputs, a, b, c, or d, is generated by the program below?

```
10 A=111  
20 B=222  
30 C=333  
40 PRINT A,B;C  
50 END
```

- a. 111 222 333
b. 111 222 333
c. 111 222 333
d. 111 222 333
4. An engineer wants to keep using the display for his output device, but needs to print a catalog of all programs on the printer addressed by select code 9. What should he type at the keyboard?
5. Write a program that:
- a. Halts and asks for the user to input a number.
 - b. Prints the number on the display screen.
 - c. Prints the number on a printer addressed by select code 26.
 - d. Stores itself as a PROG file called "PRINTER."

Handling Numbers

This lesson teaches you how to work with numbers in numeric variables. Here are the topics you'll cover in this lesson:

- Pre-run and run.
- REAL and INTEGER variables.
- Declaring variable types.
- Using DATA, READ, and RESTORE to put data into variables.
- Multiple INPUT on one line.

When it comes to doing problems in the real world, your handling of variables and data will determine just how effective your programs are.

Pre-Run and Run

When you press [RUN], it seems as though the computer executes your program immediately. In reality, though, there's a two-step process that occurs. The computer must:

- *Step 1:* Pre-run the program.
- *Step 2:* Run the program.

During pre-run, the program is laid out in memory before the actual run. When it pre-runs a program, the computer:

- Identifies the main program and any subprograms and functions.
- Checks to make sure variable labels and statements all match.
- Checks nesting of subroutines.
- Reserves computer memory for variables.

If errors are detected during pre-run, the computer gives you an error message. For instance, if your program doesn't have an END statement (a definite no-no), you'll see an error message after pre-run.

Variables

In BASIC, your computer keeps track of numbers and words by means of *variables*. A variable is actually a name; you use it to identify a number, a series of numbers, a single word, even all the words in a book!

Variable Names – How Long?

Variable names can be as long as 15 characters. The first character must be a capital letter, and all the rest lowercase. You can use any combination of letters and numbers, but the only other character you can use is the underscore line (_) or a trailing dollar sign (\$). Look at these examples:

Legal Variable Names	Illegal Variables
Location_1_a	Location 1_a
A1	1A
I	i
I_no_1	I#1
Mr_Wonderful	Mr. Wonderful
Ajax\$	AJAX
Income_1988	Income-1988

Types of Variables

There are several types of variables:

- Real numbers.
- Integer numbers.
- Strings.
- Complex numbers.

Let's look at each of these in turn.

Real: A real number is a full-precision floating-point number. It needs eight bytes of computer memory for each number stored. What this means is that real variables can be very large (up beyond 1×10^{308} !). But they use more memory, and it takes more time for the computer to get to them.

Note



Unless you tell it otherwise, your computer assumes any numeric variable is real. This guarantees maximum precision in all calculations.

Integer: This is also a number, but it's one that is rounded to the nearest whole number. (So 1.0035 is rounded to 1.) The number uses only two bytes of memory for these integers. Integer quantities can be between -32768 and $+32767$.

String: This variable isn't a number, but rather is made up of alphabetic and numeric *characters*. It can be text, a name, month, book title, or anything else you want to save and use as alphabetic information rather than numbers. A string variable has a dollar sign (\$) at the end to identify it as a string.

You'll learn more about strings in the next lesson.

Complex: Complex numbers are written as the sum of a real number and an imaginary number. (An imaginary number is any real number multiplied by the square root of -1 .) You won't use complex variables in this course.

Declaring Variable Types

Another thing you'll often want to do early in your programs is to declare variable types. You can use `INTEGER` and `REAL` to declare that a variable is a certain type. For instance, if you wanted the variables `I` and `Number` to be integers (cutting off any numbers after the decimal point), you could use this statement:

```
10 INTEGER I, Number
```

Five Fail-Safe Rules for Variables

Having trouble with variables in your programs? Remember these five simple rules for using LET or implied LET:

1. If there's just one variable, it's always on the left of the = symbol.

100 LET V=12 is correct 100 LET 12=V is wrong

2. The variable on the left of the = symbol always stands alone.

50 V=A*B is correct 50 A*B=V is wrong

3. The variable on the left of the = symbol receives a new value, equal to the number on the right.

90 LET A=4/2 This statement means A is now 2.

4. If a variable appears on both sides of the = symbol, its old value (on the right) is replaced by the new, calculated value on the left.

30 LET A=10
40 LET A=A/5 The new value of A is now 2.

5. If a variable appears only on the right of the = symbol, its value isn't changed.

10 A=5
20 B=100 The new value of B is 10. A keeps its original
30 B=A*2 value, 5.

Hint: Try reading LET statements from right to left. First determine the number on the right of the = symbol, then assign it to the variable on the left of the symbol.

Assigning Numbers to Variables in a Program

If you've worked through lesson 2 of this course, you already know one way to assign a number to a variable; it's the LET statement:

```
10 LET A=25
```

Remember, you don't need the word LET. You can simply use:

```
10 A=25
```

Another way of assigning numbers to variables in a program is with the DATA and READ statements.

The Dynamic Duo of DATA and READ

DATA and READ are used to assign numbers to variables from within a program. The READ statement "reads" whatever is in the DATA statement.

Look at the two statements together:

```
10 DATA 143
20 READ A
```

Line 10 places the number 143 in the computer's memory. The READ statement in line 20 then assigns that number to the variable A. It's just as if the computer had executed a LET statement:

```
10 A=143
```

You can combine variables in READ statements and numbers in DATA statements, like this:

```
10 DATA 11,25,143
20 READ A,B,C
```

This is the same as three LET statements:

```
10 A=11
20 B=25
30 C=143
```

The Data Stream

The computer actually puts numbers (or strings) from all DATA statements into its memory in one long stream. It does this during pre-run.

The READ statement is executed at runtime, after all data has been placed in memory. This means that it doesn't matter whether a READ is before or after a DATA. It also means that a single READ can get data from many DATA statements.

Quickly type in this mini-program:

```
SCRATCH_┘
EDIT_┘
```

```
10 DATA 1,2,3
20 READ A,B,C
30 PRINT A;B;C
40 DATA 4,5,6,7,8,9
50 READ A,B,C
60 PRINT A,B,C
70 READ A,B,C
80 PRINT A,B,C
90 END
```

Then run the mini-program. The output should look that shown below.

RUN_↓

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

What happened? At pre-run, the computer looked at all the DATA statements, and linked them to store a data stream in memory.

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

Later, at runtime, the READ statement in line 20 read the first three numbers and placed them in variables A, B, and C.

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

↑
The READ in line 20
reads to here

Line 50 read the next three quantities.

1 2 3 4 5 6 7 8 9
 ↑
 Line 50 reads to here

Line 70 read the last three:

1 2 3 4 5 6 7 8 9
 ↑
 Line 70 reads to here

The Data Pointer

When reading data, the computer marks its place with a "data pointer." There is a separate data pointer for each program segment; if execution leaves one segment, the pointer in that segment waits until execution returns. (This will be more clear after you've worked through lesson 6, which explains subroutines and subprograms.)

Restoring the Data Pointer

The RESTORE statement resets the data pointer to the beginning of the data stream. For example, add a line to the mini-program you just ran:

EDIT_↓

```
65  RESTORE 40
```

Then run the program again. Now the output looks like this:

```
1 2 3  
4     5     6  
4     5     6
```


Here's what the modified mini-program looks like:

```
10 DATA 1,2,3
20 READ A,B,C
30 PRINT A;B;C
40 DATA 4,5,6,7,8,9
50 READ A,B,C
60 PRINT A,B,C
65 RESTORE 40
70 READ A,B,C
80 PRINT A,B,C
90 END
```

During pre-run, of course, the data from lines 10 and 40 is placed in a data stream in memory. At runtime, line 20 reads the first three numbers, and line 50 reads the second three. Line 65 restores the data pointer to the beginning of the data specified in line 40, so the next READ statement begins with that data again.

If you don't use a line number or label in the RESTORE statement, the data pointer is restored to the first element in the first data statement.

Assigning Numbers from the Keyboard

LET and READ/DATA assign numbers to variables within a program. You can also assign numbers to variables from the keyboard. For this you'll use the INPUT statement.

Using the INPUT statement, you can write a single line that lets you input several different variables.

FIXER-UPPER handy person special! 2BR 1BA with materials for deck, carport, kitchen. Roof partially open for lots of fresh air. \$69,900

CUTE 3BR has pink trim, interior, exterior. Very close to freeway for easy commute. \$125,700

IDEAL STARTER HOME. Love at first sight with this 1BR beauty. Sleeps 4, if they're on good terms. Just \$105,000

COUNTRY SQUIRE. Real country living in this "ranch" house. Hear the coyotes howl, see the grass grow. 4WD vehicle a must! \$179,250

NOTRE DAME, move aside! 8BR, 4BA, features 2 spas, 6 car garage, pool, tennis, rec room. Wine cellar or in-law quarters. \$485,990

Example: Perpetually impecunious Wilkins Micawber has finally saved \$20,000—enough for a down payment on his dream house. He figures his meager salary as a clerk will let him buy as long as the payments are no more than \$1000 per month.

Micawber knows the formula for figuring the payment, P:

$$P = iA \left[\frac{(1+i)^n}{(1+i)^n - 1} \right]$$

Where

- A is the amount of the loan.
- i is the monthly interest rate.
- n is the number of payments he has to make to pay off the loan.

He knows the first thing he must do is convert the yearly interest rate and the number of payments per year to something that fits in this formula, that is:

- $i = \text{Yearly interest rate}/12/100$
- $n = \text{Years} \times 12$

Micawber has pored over the newspaper classified advertisements and found five houses of interest. This table shows the amount of the mortgage for each one, and the terms he can get from the bank for each house:

House	Amount of Loan	Interest Rate	Years to Pay
Fixer-Upper	\$49,900	11%	20
Cute 3BR	\$115,700	10.2%	25
Ideal Starter	\$85,000	10.2%	30
Country Squire	\$159,250	9.5%	35
Notre Dame	\$465,990	8.8%	40

Solution: To help Micawber, get the disk of examples that came with this course, and put it into a disk drive. Make sure you've specified that disk drive as the mass storage device, the way you learned in lesson 3. (Use the statement MSI ":",700,0" or MSI ":",1500,0".) Then load the program called "MORTGAGE" by typing:

LOAD "MORTGAGE" ↵

Now run the program to find out Micawber's monthly payment for the "Fixer-upper."

RUN ↵

Amount

When the program asks for "Amount," press keys on the keyboard to enter the cost of the home (less Micawber's minuscule \$20,000 down payment, of course):

49900 ↵

Interest rate

Enter the interest rate when you're asked for "Interest rate":

11 ↵

Years

Now enter the number of years to pay off the loan:

20 ↵

The monthly payment for a loan of \$ 49900 at a rate of 11 %
for 20 years is \$ 515.06.

The answer is \$515.06, well within Micawber's budget.

How it works: List the program, or follow along on the listing here:

LIST_1

```
10 ! MORTGAGE
20 INTEGER Years,Amount
30 INPUT "Amount",A,"Interest rate",Rate,"Years",Years
40 LET I=Rate/12/100
50 LET N=Years*12
60 LET P=I*A*(((1+I)^N)/(((1+I)^N)-1))
70 Pmt=PROUND(P,-2)
80 PRINT "The monthly payment for a loan of $";A;"at a rate of ";Rate;"%"
90 PRINT "for";Years;"years is $";Pmt
100 END
```

The key to this program is the INPUT statement in line 30. If you worked through lesson 2, you know that the INPUT statement stops the program and waits for input from the keyboard.

Line 30 is an example of a multiple INPUT statement. It waits for three variables: A, Rate, and Years.

Line 30 also shows something else about INPUT. You can display words (such as "Amount" or "Interest rate") to prompt for input.

Declaring a Variable

Micawber doesn't need to use real variables for the number of years (the variable "Years") and the amount of the loan ("Amount"). So these are declared as integer variables in line 20:

```
20 INTEGER Years,Amount
```

Rounding a Number

Micawber also doesn't need more than two decimal places of trailing numbers after his answer. So he uses line 70:

```
70 Pmt=PROUND(P,-2)
```

This means "round the contents of the variable P to two decimal places, and put the result in the variable Pmt."

Now run the program again to find the monthly payments for mortgages on the remaining houses. Start with the Fixer-Upper:

```
The monthly payment for a loan of $ 49900 at a rate of 11 %  
for 20 years is $ 515.06
```

Cute 3BR:

```
The monthly payment for a loan of $ 115700 at a rate of 10.2 %  
for 25 years is $ 1067.72
```

Ideal Starter:

```
The monthly payment for a loan of $ 85000 at a rate of 10.2 %  
for 30 years is $ 758.53
```

Country Squire:

```
The monthly payment for a loan of $ 159250 at a rate of 9.5 %  
for 35 years is $ 1308.42
```

Notre Dame:

The monthly payment for a loan of \$ 465990 at a rate of 8.8 %
for 40 years is \$ 3522.88

Hmmm. It looks like Micawber can afford only the
"Fixer-Upper" and the "Ideal Starter."

Be Careful With INTEGER

Integer variables are useful and save memory. But
sometimes they can get you in trouble.

Try this: Change line 20 to the following:

```
20 INTEGER Years, Amount, I
```

Then run the program again for – say, Notre Dame:

```
ERROR 31 IN 60 Division by 0 or X MOD 0
```

The computer rounded I to the nearest integer. Since the
number is a decimal (.09 for an interest rate of 9%), the
nearest integer is zero. This causes line 20 to attempt
division by zero, giving an error.

Review Quiz

1. In this program, what is the final value of the variable
Number?

```
10 A=5  
20 Number=A*2  
30 Number=Number^2
```

2. Which of these are legal variable names?

Result
ABC
P47
Array_in
Frank-Cheerybyle
21B
Volts_rms
Word_and_figures

3. What will be the output from this program?

```
10 READ A, B, C, D
20 PRINT A; D
30 RESTORE 10
40 DATA 43, 87, 91, 12
50 PRINT B; C
60 END
```

4. Write a program that asks a user to input his or her age, weight, and number of children. Print the data with appropriate labels.
5. Write a program that uses READ and DATA to compute and print the sines of three angles: 0 degrees, 45 degrees, and 90 degrees.

Hint: BASIC "wakes up" assuming angles are in radians. So to compute the sines of angles specified in degrees, include a DEG statement early in your program to set the computer to degrees mode. Like this:

```
10 DEG
```

Handling Words in Strings

Lesson 4 showed you how to work with *numbers* in simple variables. This lesson explains how to work with *words* in string variables.

In this lesson you'll learn:

- What a string variable is.
- Reserving memory for strings with DIM.
- Using LINPUT and INPUT to enter string data from the keyboard.
- The null string.
- Replacing words in a string with other words.
- Joining strings together.
- Reversing words with REV.
- Using only parts of a string.
- Using LEN to find length.
- Using POS to find position.
- Strings, semicolons, and spacing.
- String-number conversions: VAL, VAL\$, NUM, CHR\$.
- Converting uppercase and lowercase letters with UPC\$ and LWC\$.

Strings are important because they make your work look sharp and professional. Moreover, some kinds of programs will use strings as often as they'll use numbers. For instance, if you're controlling instruments using HP-IB, those instruments need their instructions as ASCII code—that is, as strings of characters.

What Is a String?

In BASIC, a string is text within quotation marks. It can be words, letters, or even numbers. If you worked through any of the previous lessons, you've already used the simplest form of a string:

```
10 PRINT "HELLO"
```

In line 10, the word HELLO is a string.

A single string can be any length—even as long as an entire book! But when you have a long string, it's a waste of time to type the string over and over again.

To simplify the handling of strings, we use string *variables*.

String Variables

Let's review what a string variable is: It looks like a simple numeric variable, except it has a dollar sign (\$) at the end.

Like a numeric variable name, a string variable name can be up to 15 characters long. You have to use an uppercase letter for the first character; the rest can be either lowercase letters or numbers. The only other characters you can use in a string variable are the underscore line (_) and the dollar sign.

The string itself—that is, the actual text—can be as long as you want. It must be enclosed in quotation marks, though.

Type this example and run it:

```
SCRATCH_␣  
EDIT_␣
```

```
10 LET D1$="This is a string"  
20 DISP D1$  
30 END
```

```
RUN_␣
```

When you run this mini-program, the output is:

```
This is a string
```

Remember, you need the dollar sign (\$) at the end of the variable, and you must enclose the string itself in quotation marks.

Each character in a string (including spaces) occupies one byte of computer memory.

The String and the Variable

In this example, the string *variable* is D1\$. Its *string* is "This is a string". When you run the program and line 20 asks the computer to display D1\$, it shows you the string.

And remember: a string variable name always ends with a dollar sign (\$). Any time you see a dollar sign, you can be sure you're dealing with a string.

When you make string variables, use names that make sense to you. Look at these examples of legal and illegal string variables:

These Can Be String Variables	These Can't Be String Variables
Meter\$ King_james\$ A\$ Hemingway_12\$	Meter King-james\$ A \$ 12_Hemingway\$

Assigning Strings to Variables

Once you've decided on a string variable name, the next step is to assign a string of characters to it.

The simplest way of assigning words to a string variable is with the LET statement, like this:

```
10 LET A$="HELLO"  
20 LET Hemingway$="The son also rises"
```

As with numeric variables, you don't need the keyword LET. These two statements are the same:

```
50 LET B$="GOODBYE"  
50 B$="GOODBYE"
```

So far, so good: you assign strings to variables with the LET statement, just as with numeric variables.

There's one more thing to remember about strings, though. If a string is more than 18 characters long (including spaces), you have to *reserve memory* for it before you can assign it to a string variable.

Reserving Memory

You've been working with strings that are small. If a string is large – longer than 18 characters, for example – you must reserve memory space for it before you manipulate it.

In order for your computer to use long strings, it has to know how much of its memory area the string will require. It's your responsibility to reserve this memory for a long string before you actually put any information into that string.

Although memory is automatically reserved for strings of 18 characters or less, it's better programming practice to always reserve memory for every string variable. That's because, as you will see, you can manipulate and add to your strings – they can grow and grow.



Reserve memory for all strings.

Don't worry about reserving too much memory for a string. The computer simply doesn't use the excess memory. (That memory is reserved, though, and can't be put to other uses.)

Here are statements you can use to reserve memory for strings:

DIM	Dimensions and reserves memory for strings (also for REAL number arrays).
ALLOCATE	Dimensions and allocates memory "on the fly" while a program is running. Use this in the program if you don't know how large the string will be.
COM	Reserves a common area for use by more than one part of a program.

You'll learn more about COM in part 3 of this course.

To reserve memory space, use the variable name, followed by brackets. In the brackets, place the number of characters you want to reserve for the string, like this:

```
20 DIM A$[25]
```

Line 20 reserves memory space for a 25-character string in variable A\$.

```
30 DIM Quote$[500]
```

Here, line 30 reserves enough memory for a 500-character string called Quote\$.

You can dimension several variables (strings and other types) in one statement:

```
10 DIM A$[100], Buffer$[512], Circum(500)
```

Line 10 reserves space for two strings: 100 characters for variable A\$ and 512 characters for variable Buffer\$. It also reserves memory for a REAL numeric array. (You'll learn more about arrays in lesson 8.)

What if you forget to reserve memory? First type in and run this mini-program *with* the DIM statement:

```
SCRATCH_↵  
EDIT_↵
```

```
10 DIM Eliot$[25]  
20 LET Eliot$="Let us go then, you and I"  
30 PRINT Eliot$  
40 END
```

```
RUN_↵
```

The computer prints the string Eliot\$:

```
Let us go then, you and I
```

But watch what happens if you don't reserve memory—that is, if you omit the DIM statement from the mini-program. Just "comment out" the DIM statement in line 10 by placing an exclamation point in front of it.

```
EDIT 10_↵
```

```
10 !DIM Eliot$[25]
```

The exclamation point turns this line into a remark, not a statement. So now when you run the program, the DIM isn't executed:

RUN_↵

```
ERROR 18 IN 20 String ovfl. or substring err
```

You can see that without the DIM statement, the computer gives you "string overflow" error.

The moral here is that you need to reserve memory for strings.

INPUT and LINPUT

Remember the INPUT statement. It lets you assign data to variables right from the keyboard.

You can do the same thing for strings, using either INPUT or a new statement, LINPUT. Like INPUT, the LINPUT statement stops a running program and waits for you to enter a string from the keyboard. It puts everything you type, including spaces, into the variable's storage area.

Enter and run this "friendly" little mini-program:

SCRATCH_↵

EDIT_↵

```
10 DIM A$[100]
20 LINPUT "HOW'S YOUR HEALTH?",A$
30 PRINT "GLAD TO HEAR YOUR HEALTH IS ";A$
40 END
```

RUN_↵

When you run the program, line 20 causes it to wait for you to enter a string from the keyboard:

```
HOW'S YOUR HEALTH?
```

If you're feeling OK, type:

```
FINE_
```

The computer then responds with a friendly, interested answer:

```
GLAD TO HEAR YOUR HEALTH IS FINE
```

You can also use the INPUT statement to have a program wait for multiple strings, as well as numbers. For example, look at this line:

```
110 INPUT A$,B$,X
```

Line 110 causes a running program to stop and wait for you to type in two strings and a number. You'd need to type in a string for A\$, a string for B\$, and a number for the numeric variable X.

Fun With Strings

We're going to fool around with strings now. Remember these little "games" later, though – you'll need them when you use strings in your programs.

The "Null String"

Somewhere, sometime, you're going to hear about a "null string." It's nothing, really – in fact, a null string is a string containing just that: nothing. Here's an example:

```
40 A$=""
```


The double quotation marks in line 40 assign a null string to the variable A\$.

Replacing a String

As with numeric variables, string variables contain the latest string assigned to them. Try this:

```
A$="HELLO" ↵  
B$="GOODBYE" ↵  
A$=B$ ↵  
DISP A$ ↵
```

Can you figure out the current value of A\$? That's right, it's:

```
GOODBYE
```

Just as with numeric variables, you should read from *right* to *left* when evaluating something like A\$ = B\$. The variable on the *left* of the equals sign is always the one that's changed.

Putting Words Together

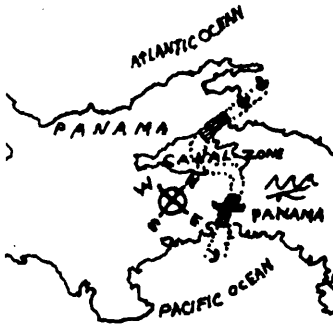
You can join strings together – the fancy word is "concatenate." Use the ampersand (&) to concatenate strings, as shown in the mini-program below:

```
SCRATCH ↵  
EDIT ↵
```

```
10 A$="A MAN"  
20 B$="A PLAN"  
30 C$=" "  
40 PRINT A$ & C$ & B$  
50 END
```

Now run the mini-program and see the results:

Reversing a String



RUN_

```
A MAN A PLAN
```

Use REV\$ to get a string that's the reverse of an existing string—that is, to reverse the order of the letters in the string.

To see an example of concatenated strings and the REV\$ function, load "PANAMA" from the examples disk (or type it in yourself using the listing below—it's not very long).

To load and run the program, type:

```
LOAD "PANAMA"_  
RUN_
```

```
A MAN A PLAN A CANAL PANAMA  
AMANAP LANAC A NALP A NAM A
```

Here's a listing of the "PANAMA" program:

```
10 ! RE-STORE "PANAMA"  
20 DIM Total$[30],Palindrome$[30]  
30 A$="A MAN"  
40 B$="A PLAN"  
50 C$=" "  
60 D$="A CANAL"  
70 E$="PANAMA"  
80 Total$=A$&C$&B$&C$&D$&C$&E$  
90 PRINT Total$  
100 Palindrome$=REV$(Total$)  
110 PRINT Palindrome$  
120 END
```

The statement in line 80 joined all the strings together and put them in the string variable Total\$. Line 90 printed Total\$ (A MAN A PLAN A CANAL PANAMA).

The REV\$ statement in line 100 reversed the order of all characters and spaces in Total\$ and assigned the new string to Palindrome\$. The order of the string was reversed by REV\$.

Note



If you look closely, you'll see that A MAN A PLAN A CANAL PANAMA is a famous palindrome—a series of letters that are the same whether read backward or forward.

Using Parts of Strings

Want to use only *part* of a string (called a "substring")? Just use the string variable and put brackets after it. Numbers inside the brackets specify which characters in the string to use.

For instance, try this exercise. Type in and run this mini-program:

SCRATCH ↵

EDIT ↵

```
10 DIM A$[22]
20 A$="USING PART OF A STRING"
30 END
```

RUN ↵

Now look at how you can use parts of strings:

A\$[2] ↵

```
SING PART OF A STRING
```

A\$[8]↵

```
ART OF A STRING
```

If you have only one number in brackets, it specifies the part of a string from that character's position in the string to the end of the string.

So A\$[2] begins with the second character in the string (reading from the left), the character "S". It continues to the end of the string. So all you see is SING PART OF A STRING, even though the actual string in A\$ is still USING PART OF A STRING.

A\$[8] begins with the eighth character in the string and continues to the end, so you see only ART OF A STRING.

You can also use *two* numbers in the brackets, separated by a comma. These specify the beginning and end – that is, the first and the last characters you want to see in the string:

A\$[2, 10]↵

```
SING PART
```

A\$[12, 13]↵

```
OF
```

The two numbers specify the first and last positions, and include all characters and spaces in between.

Another way to use just part of a string is with brackets surrounding two numbers separated by a *semicolon* (;), like this:

A\$[2;9]

In this case, the first number is the first character specified. The second number is the *number of characters* you want. For instance:

```
A$[2;9]↵
```

```
SING PART
```

The [2;9] specifies a substring beginning with the second character of A\$, and containing nine characters in all. Here's another one:

```
A$[15;7]↵
```

```
A STRIN
```

This feature is very useful in programming electronic test instruments, which usually require very detailed and specific strings for control. For instance, a voltmeter needs a string sent to it that looks like this:

```
R S V T 3
```

Function Code (F1-F4) Range Code (R1-R5)

Here's a short section of code that sets up the voltmeter for measurement:

```
80 A$="" !Null string to clear A$
90 A$[1,2]="RS" !String's 1st two characters reset meter.
100 A$[3;3]="VT3" !Next three characters set trigger mode.
110 INPUT "Enter function code (F1-F4)",A$[6,7]
120 INPUT "Enter range code (R1-R5)",A$[8,9]
```

When the operator runs this program, lines 110 and 120 stop the program and ask him or her to type in the function code and range code for the voltmeter – that is, to specify how the voltmeter is to make a measurement.

Replacing Part of a String

You can replace or change part of a string; just use brackets. Try this one:

```
SCRATCH_↓  
EDIT_↓
```

```
10 A$="WIDESPREAD"  
20 PRINT A$  
30 A$[5]=" OPEN"  
40 PRINT A$  
50 END
```

```
RUN_↓
```

The program prints the A\$ you specified in line 10, then the new, changed A\$ you specified in line 30:

```
WIDESPREAD  
WIDE OPEN
```

The statement A\$[5]=" OPEN" in line 30 replaced everything in A\$ from position 5 to the end of the string with " OPEN".

If you leave unassigned spaces before the last character of a string, they're filled with blanks. Unassigned positions after the last assigned character are undefined and ignored.

Finding the Length of a String

The LEN statement gives the length of a string. Try this example:

```
SCRATCH_␣  
EDIT_␣
```

```
10 DIM A$[100]  
20 A$="Able was I ere I saw Elba"  
30 X=LEN(A$)  
40 DISP X  
50 END
```

```
RUN_␣
```

Line 30 puts the length of A\$ into the variable X; then line 40 prints X:

```
25
```

Notice that the length of the string computed by LEN is not the same as the memory reserved for it by the DIM statement.

Finding Position Within a String

The POS function gives you the position of a substring within a string, like this:

```
POS (String1$, String2$)
```

This statement finds the beginning position of String2\$ in String1\$.

Here's a handy mini-program that uses POS to deliver the number of any capital letter in the alphabet:

SCRATCH.␣
EDIT.␣

```
10 DIM Alpha$(26)
20 Alpha$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 INPUT "Type in a capital letter",Letter$
40 X=POS(Alpha$,Letter$)
50 PRINT "The position of the letter ";Letter$;" is";X
60 END
```

When you run the program, it asks for you to type in a capital letter, then prints the position of that letter in the alphabet. For example, to find the position of the letter "M", type:

RUN.␣

Type in a capital letter

M.␣

The position of the letter M is 13

Strings, Semicolons, and Spacing

As you know, you need a separator for items in your PRINT or DISP statements; you can use a semicolon or a comma. Remember the difference?

If you use a semicolon to separate items in a PRINT or DISP statement, they appear close together. A comma moves them apart.

The semicolon can help make your output look prettier – but it works slightly differently with strings than it does with numbers. If you've just run the mini-program above, you can try this:

```
PRINT "The current X is";X;"truly"
```

Here, the semicolons before and after the numeric variable X leave a nice space around it:

```
The current X is 13 truly
```

Now modify your PRINT statement (the [RECALL] key makes it easy) to try a leading and trailing semicolon with a *string* variable:

```
PRINT "The current Alpha$ is";Alpha$;"truly"
```

This time, there are no spaces before or after the string:

```
The current Alpha$ isABCDEFGHIJKLMNOPQRSTUVWXYZtruly.
```

In its wisdom, HP BASIC knows that you probably don't want to run numbers together, but that you *do* want complete control over how words appear. Just remember to add spaces as needed when you PRINT or DISP strings.

Conversions

As you know, the computer sees all strings as words, not numbers—you can't do calculations with them. But there are times in programming when you'll need to convert strings to numeric data, or vice versa. Luckily, HP BASIC has a host of functions you can use for this purpose.

Converting Strings to Numbers

Suppose you had a list of names and account balances, all in strings, like this:

```
A1$="Heep, U; $152.40"
```

Right now you can't do calculations (such as addition) with that \$152.40 because it's still *string* data. To convert it to a number, you can use VAL.

The VAL function converts a number that's in a string to something the computer can use for calculations. Try this mini-program to compute a 20% discount for U. Heep's account:

```
SCRATCH_␣  
EDIT_␣
```

```
10 A1$="Heep, U; $152.40"  
20 X=VAL(A1$[11])  
30 X=.2*X  
40 PRINT X  
50 END
```

When you run this mini-program, it gives you the discount:

RUN_↓

30.48

How it works: The entire A1\$ variable, including the characters "152.40", is a string. Line 20 uses VAL to find the numerical value of the item beginning in position 11 of the string A1\$. Since 152.40 begins in position 11, the program puts that value in variable X, then calculates and prints 20% of the value.

The first character in the substring to be handled by VAL *must* be a digit, a plus or minus sign, decimal point, or a space. The remaining characters have to be digits, a decimal point, the sign of E, or an E (which is interpreted as an exponent of 10).

Characters to Numbers

If you need to convert a *character* in a string to a *number*, use NUM. This function converts the first character of a string to its corresponding ASCII code value.

What's an ASCII code? It's just a number. (ASCII is the *American Standard Code for Information Interchange*.) In a computer, each character – whether it's a number, letter, or symbol – can be represented by a seven-bit ASCII code.

NUM returns the ASCII code value of its argument. Try this mini-program:

SCRATCH_↓

EDIT_↓

```
10 A=NUM("A")
20 DISP A
30 END
```

RUN_↓

65

The decimal value 65 is an ASCII character code – that is, it's the ASCII value assigned to that character. In the ASCII code, uppercase and lowercase letters have different numbers assigned to them.

Numbers to Characters

Use CHR\$ to convert ASCII codes (numeric values) to individual characters.

Try this:

CHR\$(35) ↓

#

The CHR\$ function assumes the number in parentheses is an ASCII code and produces the character associated with that number. You can think of CHR\$ as the "reverse" of NUM.

CHR\$ gives you more programming flexibility.

Inserting Quotation Marks

There's a special problem in using quotation marks within a string. If you try to insert quotation marks, the computer sees them as the beginning or end of the string. So what you must do is type *double* quote marks to produce a single quotation mark within the string.

Try this; type:

PRINT ""Come in, come in,"" he leered."

The computer displays:

```
"Come in, come in," he leered.
```

The double quotation marks cause a single quotation mark to be printed.

Lowercase and Uppercase Conversions

You can use UPC\$ and LWC\$ to convert a string to all uppercase or all lowercase characters. Try this one:

```
PRINT UPC$("UP down UP down"),
```

```
UP DOWN UP DOWN
```

The UPC\$ function converts the string "UP down UP down" to all uppercase letters.

Uppercase and lowercase letters have different ASCII values. (An uppercase "A" is 65, for instance, while a lowercase "a" is 97.) But you can use UPC\$ or LWC\$ to make sure a program processes either character the same way. Look at this code:

```
.  
.   
100 LINPUT A$  
110 A$=UPC$(A$[1,1])  
.   
.
```

No matter what the user of this program types for A\$ at the LINPUT statement, line 110 converts it into the uppercase first letter of the word. So the user could type Yes, yes, Y, or y, and the program would evaluate A\$ the same way.

Useful String Functions

Here are some of the most helpful functions to help you evaluate and manipulate strings in your programs:

Function	What It Does
CHR\$ (N)	Converts the numeric value N into an ASCII character.
DVAL (S\$,B)	Returns the whole number value of S\$ using base B. B describes the base of the number and must be 2, 8, 10, or 16.
DVAL\$ (N,B)	Returns the string equivalent of whole number N using base B. B must be 2, 8, 10, or 16.
IVAL (S\$,B)	Returns the integer value of S\$ using base B. B must be 2, 8, 10, or 16.
IVAL\$ (N,B)	Returns the string equivalent of integer N using base B. B must be 2, 8, 10, or 16.
LEN (S\$)	Returns the number of characters in the string S\$.
LWC\$ (S\$)	Replaces all uppercase characters in S\$ with lowercase characters.
NUM (S\$)	Returns the decimal value of the first character in string S\$.
POS (S\$,T\$)	Returns the beginning position of T\$ in S\$.
REV\$ (S\$)	Returns a string that is the reverse of string S\$

Function	What It Does
RPT\$ (S\$,N)	Returns a string that contains N repetitions of string S\$.
TRIM\$ (S\$)	Strips all leading and trailing blanks from S\$ and returns result.
UPC\$ (S\$)	Replaces all lowercase characters in S\$ with uppercase characters.
VAL (S\$)	Converts string S\$ into a numeric value; used to convert ASCII numbers to real values.
VAL\$ (N)	Converts numeric value N into a string expression.

Review Quiz

1. Which of these are *not* legal names for string variables?

Spentlow\$
A\$5
Volts
Volume_12\$
ZONE\$
7M\$
Anagram\$
Taxpayer_43\$

2. How would you set the variable N\$ equal to an "N" *without* using quotation marks?

3. In this section of code, what's wrong with line 20?

```
10 A$="Varden, 1102905"  
20 A= VAL (A$)
```

4. What will be the result if you run this program?

```
10 READ Meterstring$  
20 PRINT Meterstring$  
30 A$= "1103295"  
40 PRINT "The new input is "; A$  
50 DATA "JCX3428911MLP"  
60 Meterstring$= Meterstring$&A$  
70 PRINT Meterstring$  
80 END
```


- 5.** Write a "string evaluator" program. It should:
 - a.** Accept a string of up to 100 characters from the keyboard.
 - b.** Print the string.
 - c.** Tell how many characters are in the string.
 - d.** Print the inverse of the string without leading or trailing spaces.

Decisions, Decisions

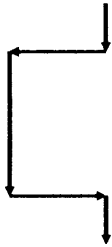
This is a very important lesson. In it, you will learn how your programs can make decisions – something that turns your computer from a large, comparatively expensive calculator into a "thinking" machine.

You will learn about these topics:

- GOTO.
- Subroutines: GOSUB and RETURN.
- Subprograms: CALL and SUBEND.
- Decision-Making: IF-THEN-ELSE, ON, SELECT CASE.
- Comparisons.

The GOTO Statement

When a GOTO statement is executed in a running program, it causes the program to jump immediately to the specified line number (or line label – more about these in a moment). Execution then resumes at the new location. This jump is called a "branch."



```
180 R=R+2
190 Area=PI*R^2
200 GOTO 240
210 Width=Width+1
220 Length=Length+1
230 Area=Width*Length
240 PRINT Area
250 !
```

When line 200 is executed, the program jumps (branches) immediately to line 240. It skips lines 210-230. The value for Area that's printed is $PI * R^2$, *not* $Width * Length$.

GOTO used in this way is called an *unconditional branch*, because it always branches execution, no matter what the condition of any variable.

GOTO can put a program into an endless loop. Type and run this mini-program:

```
SCRATCH_␣
EDIT_␣
```

```
10 A=A+1
20 PRINT A
30 WAIT .1
40 GOTO 10
50 END
```

```
RUN_␣
```

Every time line 40 is executed, execution branches back to line 10. The variable A is incremented (that is, the value 1 is added to it) and the new value for A is printed. Then the loop is repeated.

This endless loop will continue executing forever, or until you turn off the computer or press [STOP] or [RESET].

Changed Line Numbers

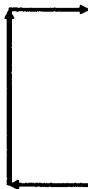
What happens if you change a program's line numbers with the REN command? No problem — all your GOTO statements and other branches are renumbered automatically as well.

Line Labels

Besides line numbers, HP BASIC also uses *line labels* for branches. A line label looks like a variable, except it has a colon (:) at the end and is always followed by a statement. The simplest statement is an exclamation point (a remark) after a line label, like this:

```
210 Print_routine: !
```

When the GOTO statement is executed, it causes branching to the line label.



```
210 Print_routine: !  
220  
230  
240  
250 GOTO Print_routine
```

Like variables, line labels can be up to 15 characters long. The first letter is always uppercase, and the rest are lowercase. You can use numbers or letters in a label; the only other character that's legal is the underscore line (_).

The line label can include statements besides the exclamation point, of course. These are examples of line labels in a program:

```
120 Print_routine: PRINT "Waltz step"
200 Task_1: !
90 Calculate: A=PI*R^2
```

Forget GOTO!

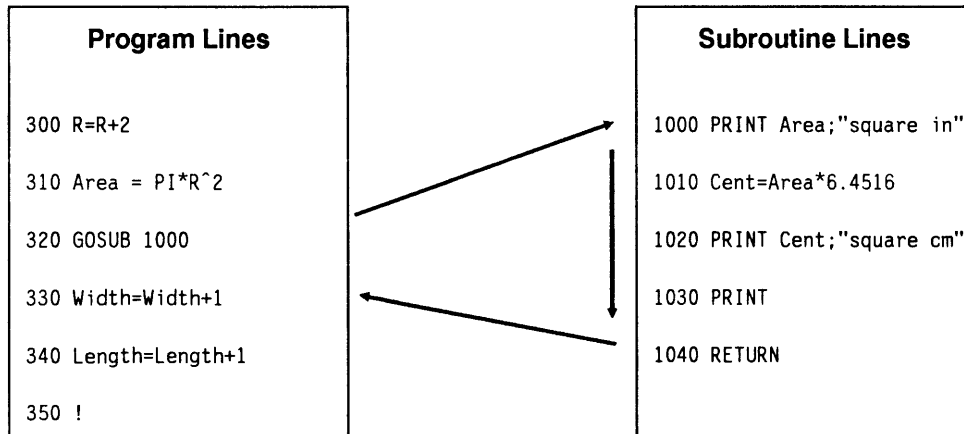
Now that you've learned the unconditional GOTO statement, forget it! If you design your programs well, you should *never* need GOTO. There are other statements for branching that are much better, including GOSUB, CALL, and FN. You'll soon learn how to use them.

Cardinal Rule

Don't use unconditional GOTOs.

Subroutines

Using GOSUB to branch to a subroutine is a much better way than GOTO to do an unconditional branch. When a running program encounters a GOSUB statement, it transfers execution to the specified label or line number. It then executes until it encounters a RETURN statement.



The RETURN after a GOSUB call causes execution to "return" to the main program and continue.

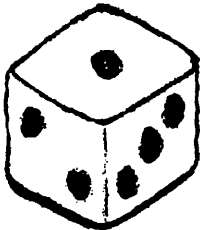
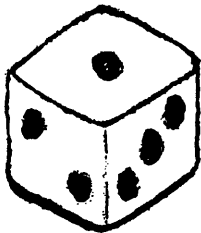
Line 310 calculates a value for the variable Area. Line 320 is the GOSUB call; it transfers execution to line 1000 which, of course, is in a different part of the program.

The print routine in lines 1000-1030 prints the value of area in square inches and in square centimeters.

When line 1040 is executed, the RETURN statement transfers execution back to the main program. Execution then continues with line 330, line 340, and so on.

The same subroutine may be called from many different places in a program.

Example: The fun-loving Alfred Jingle loves to gamble — especially rolling those dice. He's recently gone electronic, though, with his program called `ROLL_DICE`. If you dare to play with Jingle, load the program `ROLL_DICE` from the disk of examples:



```
LOAD "ROLL_DICE" _
```

When you press RUN, the program simulates the roll of a pair of dice. It prints two numbers from 1 to 6:

```
RUN _
```

```
DIE IS 2  
DIE IS 3
```

To see how it works, list the program:

LIST_↓

```
10 !RE-STORE "ROLL_DICE"  
20 !  
30 CLEAR SCREEN  
40 PRINTER IS 1  
50 GOSUB Roll_one_die  
60 GOSUB Roll_one_die  
70 STOP  
80 Roll_one_die: !  
90 RANDOMIZE  
100 Die=INT(RND*6)+1  
110 PRINT "DIE IS"; Die  
120 RETURN  
130 END
```

When you run ROLL_DICE, the statement GOSUB Roll_one_die in line 50 causes a branch to that label (in line 80).

In the subroutine, RANDOMIZE in line 90 creates a random seed for use by RND, the random number generator in line 100. The statement Die = INT(RND*6) + 1 gives an offset that keeps the value for Die a number from 1 through 6, just like a real die.

After the value for one die is printed by line 110, the RETURN in line 120 sends execution back to the main program. Then the subroutine is called again by line 60, and it calculates and prints the value for another die.

Since there are two dice, this is a perfect use for a single subroutine that is called twice.

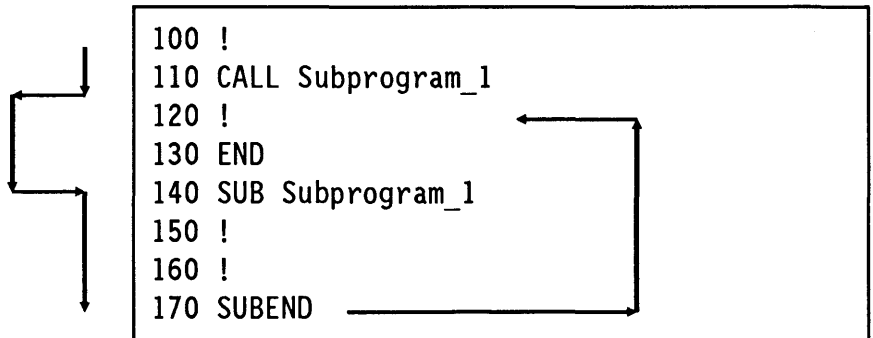
A few points about subroutines:

- The subroutine and its RETURN must be before the program's END statement.
- Branching to a subroutine "freezes" some parts of execution, such as a data pointer being used by a READ statement. When execution returns to the calling program, the "frozen" pointer or other element continues as before.
- All variables in subroutines are *global*; they have the same meaning in the subroutine and in the main program.

Subprograms

A *subprogram* is another way you can transfer execution temporarily and return. A subprogram is like a "program within a program," so you treat it as you would any program, including reserving memory and declaring variables at the beginning.

A subprogram is separate from the main program; in fact, it's after the END statement. But when it's called from the main program, execution returns to the main program after the call just like with a subroutine.



To use a subprogram, insert a `CALL` statement in your main program.

Subprogram Components

The first line in a subprogram always begins with `SUB` followed by the subprogram's name. The last line is always `SUBEND`. The `SUBEND` statement acts like a `RETURN` for a subroutine—it returns execution to the calling program.

(Incidentally, `CALL` can call a subprogram or a *function*. Functions are like subprograms, in that they are placed after the end of the main program. You'll learn about functions later on, in lesson 7.)

The keyword `CALL`, like `LET`, is optional. So these two statements accomplish exactly the same thing:

```
450 CALL Home
460 Home
```

Both line 450 and line 460 branch execution to the subprogram named "Home".

To see how subprograms are used, load the program `SUBPR_DICE`. (It's another dice-playing program for Alfred Jingle.)

```
LOAD "SUBPR_DICE" ↵
LIST ↵
```

```
10 !RE-STORE "SUBPR_DICE"
20 !
30 CLEAR SCREEN
40 PRINTER IS 1
50 CALL Roll_one_die
60 CALL Roll_one_die
70 END
80 SUB Roll_one_die
90 RANDOMIZE
100 Die=INT(RND*6)+1
110 PRINT "DIE IS";Die
120 SUBEND
```

When you run the program, the results for you (or Jingle) should be the same.

A few words about subprograms:

- Subprograms begin with SUB and end with SUBEND.
- Subprograms always come *after* the main program's END statement.
- Subprograms have *local* variables. Their usage and meaning may be different in other subprograms or in the main program.

Why Use Subprograms?

Because they're appended after the end of the program, subprograms have advantages over subroutines and other structures. For instance, you can write a program that appends a subprogram from a disk, runs the subprogram, then wipes it out and appends another subprogram, and so on. You'll learn some of these programming techniques when you reach part 3 of this course.

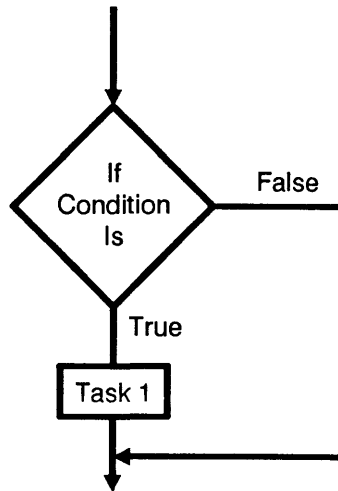
Making Decisions

One of the most useful features of any computer is its ability to make decisions based on facts.

In BASIC, the computer asks the question "Is such-and-such true?" Then execution branches or not, depending on the answer to the question. For example, if A is greater than B, execution may branch to another part of the program. If B is greater than A, execution continues uninterrupted.

IF-THEN

The simplest branch decision-making statement in BASIC is IF-THEN: *if* a condition is true, *then* the program prints, or branches, or shoots off fireworks. If the condition is not true, execution continues unaffected.



In this example, if the condition is true, Task 1 is executed. If the condition is false, it's not executed.

Here's how you could accomplish this with IF-THEN:

```
50 IF A=1 THEN GOSUB Task_1
60 !
70 !
80 STOP
90 !
100 Task_1: !
110 PRINT A
110 RETURN
```

Comparisons

How do we make comparisons? One way is with the equals sign:

```
100 IF A=0 THEN PRINT "IT'S NOTHING"
```

You can also find out if one variable is greater or less than another:

```
200 IF A<0 THEN PRINT "IT'S BIG"
```

In line 200, if A is less than zero, then the message is printed.

Combinations are also allowed:

```
350 IF A>=B THEN
```

If, when line 350 is executed the variable A is greater than or equal to B, then some action is taken.

You can compare strings, too:

```
570 IF A$=B$ THEN GOSUB Same_name
```

Here, if the characters in string A\$ exactly match those in string B\$, then the program branches to subroutine Same_name.

Here are the comparisons you can use:

A > B	If A is greater than B...
A < B	If A is less than B...
A > = B	If A is greater than or equal to B...
A < = B	If A is less than or equal to B...
A = B	If A equals B...
A < > B	If A is not equal to B...

Be Careful With Comparisons

You must be careful when comparing for equality. For instance, try this comparison:

```
SCRATCH_┘  
EDIT_┘
```

```
10 A=32  
20 B=2^5  
30 IF A=B THEN PRINT "EQUAL"  
40 END
```

Now run the program. The variables should be equal, since we know that 2 to the 5th power (that is, 2 x 2 x 2 x 2 x 2) is the same as 32.

RUN_1

What happened? They should be equal. But because of a rounding error in the computer, 2^5 as calculated is not *exactly* the same as 32. It's the same for all practical purposes, but it's not exact. So line 30 is not true, and the program continues to the END statement without printing.

To get around this problem, you could use a range of values, like this:

```
30 IF B<=32.1 AND B>= 31.9 THEN PRINT "EQUAL"
```

Or you could use the math function ROUND to round the calculated number, like this:

```
30 IF A=DROUND(B,2) THEN PRINT "EQUAL"
```

Now line 30 rounds B to two decimal places, then compares it to A. This result is also true.



Don't compare real numbers for exact equality. Use a range instead, or round the numbers.

IF-THEN with AND-OR

You can base an IF-THEN decision on more than one choice, using AND or OR. Try this:

SCRATCH ↵

EDIT ↵

```
10 INTEGER A,B
20 A=3
30 B=4
40 IF A=3 OR B=3 THEN PRINT "CONDITIONS MET"
50 PRINT "ENDING PROGRAM"
60 END
```

RUN ↵

Since one condition ($A = 3$) is met, the program prints:

```
CONDITIONS MET
ENDING PROGRAM
```

If you use AND instead of OR in the IF-THEN statement, both conditions must be true for THEN to be executed. Change line 40 to IF-AND-THEN:

EDIT 40 ↵

```
40 IF A=3 AND B=3 THEN PRINT "CONDITIONS MET" ↵
```

Now see what happens when you run the program:

RUN ↵

```
ENDING PROGRAM
```

Since both conditions *aren't* true now, execution "falls through" to line 50 without printing in line 40.

IF-THEN with END IF

When you have several steps to perform after THEN, use IF-THEN with an END IF statement later, like this:

```
580 IF A=0 THEN
590 A=B
600 PRINT "A is zero, so A=B"
610 END IF
620 !Program continues
630 !
```

In line 580, if A equals zero, everything after it to line 610 is executed. If in line 580 A is *not* equal to zero, execution skips immediately past the END IF statement and proceeds with line 620.

IF-THEN with END IF and ELSE

When you have *two* long alternatives, you can use an IF-THEN statement with ELSE and END IF like this:

```
100 IF A=1 THEN
110
120
130
140 ELSE
150
160
170 END IF
180
190
```

If A = 1, this is executed

If A not equal to 1, this is executed

Execution "falls through" and continues

In this case, if A equals 1, everything between THEN and ELSE (that is, lines 110-130) is executed.

If A is not equal to 1, execution skips to the ELSE statement in line 140 and executes lines 150-160.

No matter whether THEN or ELSE is executed, the program eventually "falls through" to the END IF statement in line 170.

Here's how to choose which IF-THEN to use:

- To perform one task or a subroutine call, then continue, use IF-THEN:

```
160 IF Pointer >=1 THEN Pointer=1
```

- To perform more than one task after THEN, use IF-THEN and END IF:

```
100 IF Ph>7.7 THEN
110     PRINT "Final Ph=";Ph
120     GOSUB Next_tube
130 END IF
140 ! Program continues here
```

- To choose one of two long alternatives, use IF-THEN and ELSE, followed by END IF:

```
40 IF X <=0 THEN
50 BEEP
60 DISP "Improper argument"
70 ELSE
80 Root=SQRT(X)
90 END IF
```

Now it's time to try an example.

Example: Like most algebra students, Barnaby Drudge is terrified of quadratic equations of the form $Ax^2 + Bx + C = 0$. Even with his computer, it's tough for him to figure out the two roots:

$$R1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

$$R2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

He does have a secret weapon: it's the QUADRATIC program on the disk of example programs. Help Barnaby by loading the program now and running it to find the roots of this equation:

$$x^2 + x - 6 = 0$$

LOAD "QUADRATIC" ↵

RUN ↵

The value for A is

1 ↵

The value for B is

1 ↵

The value for C is

-6 ↵

```
R1= 2
R2=-3
```

Now list the program to see how it works:

LIST_↓

```
10 !RE-STORE "QUADRATIC"
20 PRINTER IS 1
30 CLEAR SCREEN
40 PRINT "To find the two roots, enter"
50 PRINT "the values for A, B, and C."
60 INPUT "The value for A is",A
70 INPUT "The value for B is",B
80 INPUT "The value for C is",C
90 D=B^2-4*A*C
100 GOSUB Root_1
110 GOSUB Root_2
120 STOP
130 Root_1: !
140 R1=(-B+SQR(D))/(2*A)
150 PRINT "R1= ";R1
160 RETURN
170 Root_2: !
180 R2=(-B-SQR(D))/(2*A)
190 PRINT "R2= ";R2
200 RETURN
210 END
```

Barnaby is happy with his program – as far as it goes. But he still has problems with some conditions:

1. What if the quantity $B^2 - 4AC$ is a negative number? (The program won't compute the square root of a negative number.)
2. What if $A = 0$? (The equation isn't a quadratic at all.)

Barnaby is tearing his hair. But you'll fix these problems and put Barnaby at his ease.

To handle the problem if $B^2 - 4AC$ is a negative number, add this statement:

```
91 IF D<0 THEN PRINT "CAN'T COMPUTE COMPLEX ROOTS!"
```

To take care of a condition when $A = 0$, add this code:

```
92 IF A=0 THEN  
93 PRINT "A=0. NOT A QUADRATIC!"  
94 GOTO 40  
95 ELSE
```

Also add an END IF to go along with the IF-THEN statement in line 92. Add the END IF as line 115, before the STOP statement:

```
115 END IF
```

When you're done, list the modified program. It should look like the one on the following page.

```
[PAUSE]  
LIST_↓
```

```

10 !RE-STORE "QUADRATIC"
20 PRINTER IS 1
30 CLEAR SCREEN
40 PRINT "To find the two roots, enter"
50 PRINT "the values for A, B, and C."
60 INPUT "The value for A is",A
70 INPUT "The value for B is",B
80 INPUT "The value for C is",C
90 D=B^2-4*A*C
91 IF D<0 THEN PRINT "CAN'T COMPUTE COMPLEX ROOTS!"
92 IF A=0 THEN
93 PRINT "A=0. NOT A QUADRATIC!"
94 GOTO 40
95 ELSE
100 GOSUB Root_1
110 GOSUB Root_2
115 END IF
120 STOP
130 Root_1:!
140 R1=(-B+SQR(D))/(2*A)
150 PRINT "R1= ";R1
160 RETURN
170 Root_2:!
180 R2=(-B-SQR(D))/(2*A)
190 PRINT "R2= ";R2
200 RETURN
210 END

```

Now try the modified program for the equation
 $x^2 + 2x + 2 = 0$:

RUN ↵

The value for A is

1 ↵

The value for B is

2 ↵

The value for C is

2 ↵

This generates an error, as well as the message:

CAN'T COMPUTE COMPLEX ROOTS!

Now try it for $A=0$:

RUN ↵

0 ↵

1 ↵

2 ↵

You see that you've "trapped" this error, too, for Barnaby:

A=0. NOT A QUADRATIC!

If A is not zero, the program skips lines 93 and 94 and goes right to the ELSE statement in line 95. It executes all statements after ELSE until it comes to END IF in line 115.

Now that Barnaby is smiling, you can move on to the next topic.

Figurative Flowcharts

No programming course worth its salt would be complete without a discussion of flowcharts. These collections of boxes, triangles, circles, and arrows are often a much better way to see how a program works than are code listings.

If you've been programming without flowcharts, there's no need to fall into a frenzy of self-flagellation over it. But as your programs grow and become more complex, you may want to use flowcharting techniques, both to help write your programs, and to document them for others after you've gotten them to work.

There are really only two rules to remember:

1. Flow is generally from top to bottom, left to right.
2. A diamond means a decision.

Besides these two (more or less) hard-and-fast rules, there are some other conventions followed by many programmers.

Circle: The beginning or end of a program.

Parallelogram: Input or output.

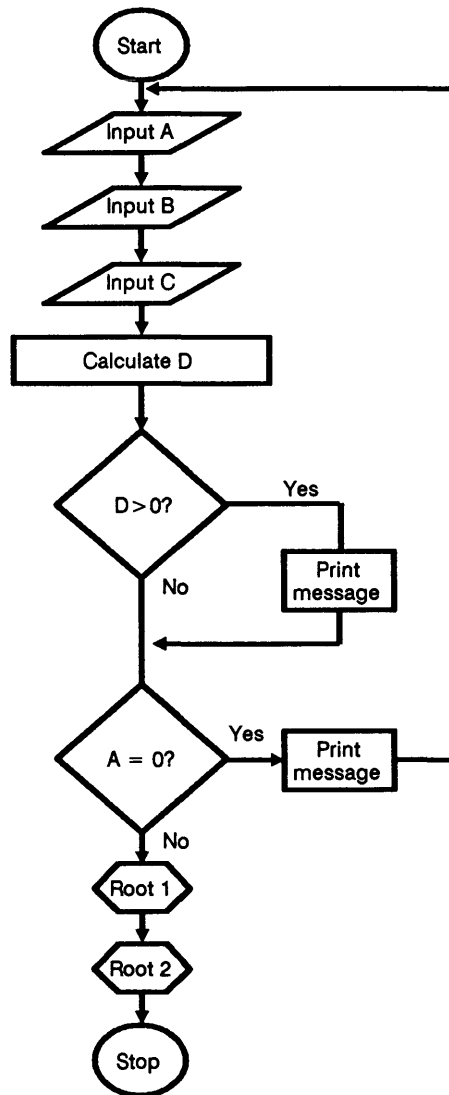
Rectangle: Computation.

Hexagon: Function or subroutine.

You should know that these are customs only, more honored in the breach than in the observance. In the real world, you'll see all kinds of flowcharts.

You can use flowcharts to help write the actual code for a program. To begin, you draw a chart that shows the major flow of a program. (The facing page is an example: it shows a flowchart for the modified QUADRATIC program.) Then you just follow the chart when you begin writing the code.

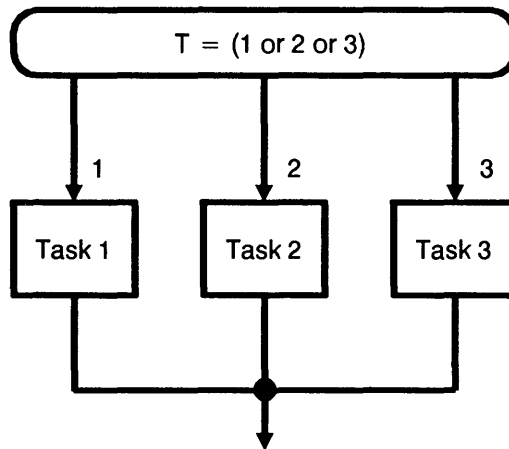
Flowchart for Modified QUADRATIC Program



Using ON

You know how to make a decision for one of two paths using IF-THEN. But what if you need to provide *several* alternatives?

If the alternatives can be specified in terms of integer values, the ON statement is good for this purpose. ON lets you choose from a list of tasks to do. Here's an example:



```
50 ON T GOSUB Task1, Task2, Task3
```

In line 50, if the variable T is equal to 1, execution transfers to the first label in the list; that is, to Task1. If T=2, the second line label in the list (Task2) is executed. When T=3, Task3 is executed.

You can use ON to specify either labels or line numbers.



Example: Although he's sailed the seven seas, Captain Cuttle of the merchant marine isn't much of a linguist. So he's written a program to help him greet people around the world. If you want to see it, load the program MORNING from the disk of examples.

```
LOAD "MORNING" _  
RUN _
```

```
To select a language, type a number:  
English--1  
French--2  
German--3  
Spanish--4  
Japanese--5  
Chinese--6
```

This program lets you type a number to select a language, then shows you how to say "Good morning!" in that language. Try it for German:

```
3 _
```

```
If you choose 3 say  
GUTEN MORGEN!
```

What about Japanese?

```
RUN _  
5 _
```

```
If you choose 5 say  
OHAYO GOZAIMASU!
```

How it works: List the program to see how it works:

LIST↵

```
1  ! RE-STORE "MORNING"
10 INTEGER Number
20 PRINT
30 PRINT "To select a language, type a number:"
40 PRINT "English--1"
50 PRINT "French--2"
60 PRINT "German--3"
70 PRINT "Spanish--4"
80 PRINT "Japanese--5"
90 PRINT "Chinese--6"
100 PRINT
110 INPUT Number
120 PRINT "If you choose";Number;"say"
130 ON Number GOSUB English, French, German, Spanish, Japanese, Chinese
140 STOP
150 English: PRINT "GOOD MORNING!"
160 RETURN
170 French: PRINT "BONJOUR!"
180 RETURN
190 German: PRINT "GUTEN MORGEN!"
200 RETURN
210 Spanish: PRINT "BUENOS DIAS!"
220 RETURN
230 Japanese: PRINT "OHAYO GOZAIMASU!"
240 RETURN
250 Chinese: PRINT "NI HAO MA!"
260 RETURN
270 END
```

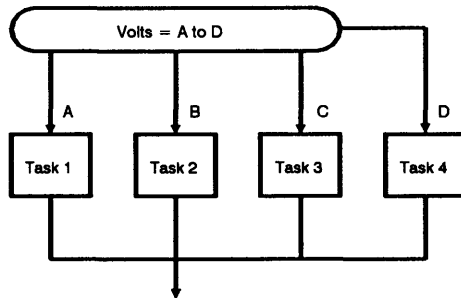
The INPUT statement in line 110 stops the program and waits for you or Captain Cuttle to type the number of the desired language.

Then the ON statement in line 130 branches execution to one of the subroutines based on the number you typed. So when you input the number 3, line 130 branches execution to the third subroutine (line label) in the list – that is, to the subroutine labeled German. This subroutine prints the words for a hearty "GUTEN MORGEN!"

SELECT-CASE

Suppose you want to choose from among many tasks, but can't use ON because you can't guarantee that the variable is an integer. What then?

One solution is the SELECT statement, followed by several CASE statements.



```
130 SELECT Volts
140 CASE <= 1
150 GOSUB Task1
160 CASE = 2
170 GOSUB Task2
180 CASE >= 5
190 GOSUB Task3
200 CASE ELSE
210 GOSUB Task4
220 END SELECT
230 !
```

In the example above, line 130 examines the current value for the variable Volts. Then the SELECT statement compares the value to each of the CASEs below it. If a CASE is true, the next statements are executed; if false, the program skips to the next case.

In the example, a different subroutine is executed, depending on the case:

- If Volts is less than or equal to 1 (line 140), execution is branched to the subroutine Task1.
- If Volts is equal to 2, subroutine Task2 is executed.
- If Volts is greater than or equal to 5, Task3 is executed.
- If Volts doesn't match one of the other cases, Task4 is executed.

So if Volts = 6, the SELECT statement would look at all the cases, with these results:

```
140 CASE<=1
```

Line 140 is not true.

```
160 CASE=2
```

Line 160 is not true.

```
180 CASE>=5
```

Line 180 is true, so subroutine Task3 is executed.

```
200 CASE ELSE
```

Line 200 is not true.

```
220 END SELECT
```

When there are no more cases, line 220 continues execution.

In your programs, CASE ELSE should take care of any condition that doesn't fit one of the other cases.

Incidentally, it's usually a much better programming practice to use SELECT-CASE instead of ON-GOTO.

Review Quiz

1. Where will execution halt for this section of code: line 60 or line 70?

```
10 A=1
20 B=2
30 C=3
40 D=4
50 IF A<B AND C<=D-2 THEN 70
60 STOP
70 END
```

2. What's wrong with this section of a mini-program?

```
10 GOSUB Init
20 GOSUB Print
30 STOP
40 Init:!  
50 DIM A$(40)
60 INTEGER I
70 RETURN
80 Print:!  
90 PRINT "Printing"
100 SUBEND
110 END
```

3. Remember the MORNING program? Modify it to use SELECT-CASE instead of ON.

Repetition, Repetition

Repeating one task over and over again, perhaps with a small change to a variable each time, is another fundamental tool of any programming language. HP BASIC gives you a number of ways to repeat a task.

In this lesson, you'll learn about:

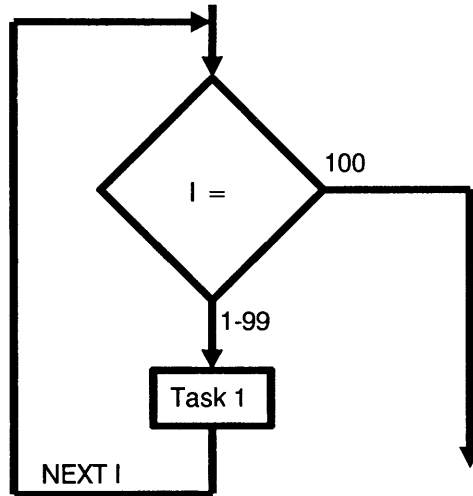
- Loops: FOR-NEXT, REPEAT UNTIL, WHILE, LOOP-EXIT IF.
- INDENT
- The live keyboard.
- FN functions.

The FOR-NEXT Loop

If you worked through lesson 3 earlier, you already met the FOR-NEXT loop. To refresh your memory, here's how one looks:

```
70  FOR I=1 TO 50
80  GOSUB Task1
90  NEXT I
100 !
```

The first time program execution reaches line 70, it sets the variable I equal to 1. Then it executes subroutine Task1 and returns.



The NEXT I statement in line 90 sends execution back to line 70 again, where the value of I is increased to 2, Task1 is executed again, and the NEXT I statement again returns execution to line 50.

This is called a "loop." The program stays in this loop until I equals 50. Then execution continues with the next line of the program after the loop – that is, with line 100.

Specifying a STEP

In the example above, I is the "loop counter." If you don't want to change a loop counter by 1 each time, you can specify a STEP. Try this example:

SCRATCH_┘

EDIT_┘

```

10 FOR J=1 TO 150 STEP 5
20 PRINT J
30 WAIT .1
40 NEXT J
50 END

```

When you run the program, you can see that the loop counter, J, changes not by one, but by five each time through the loop:

RUN_↓

```
1
6
11
.
.
141
146
```

The program exits the loop when the value for J in line 10 becomes 151 – before the PRINT statement in line 20.

Negative Step

Another way to use STEP is to specify a *negative* step interval; this way, the loop counts down.

To see a negative step in action, modify the mini-program so it looks like this:

EDIT 10_↓

```
10 FOR J=150 TO 1 STEP -1
20 PRINT J
30 WAIT .1
40 NEXT J
50 END
```

RUN_↓

You can see the program counts down from 150 to 1:

```
150
149
148
147
```

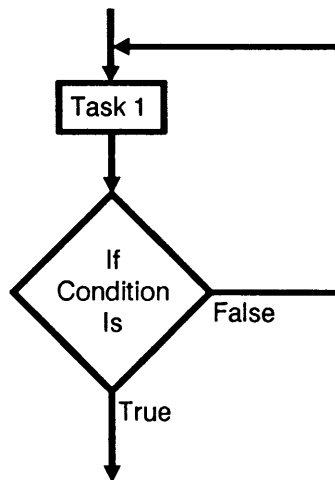
When do you use STEP? When can you omit it from your FOR statements? Luckily, there's a rule on the subject:

Cardinal Rule 

Unless the step interval is +1, you always need to specify STEP.

REPEAT and UNTIL

The FOR-NEXT loop executes until its counter reaches a specified number. Another way of creating a loop is by using REPEAT and UNTIL. This loop *repeats* over and over again *until* a specified condition occurs.



```
50 REPEAT
60 GOSUB Task1
70 UNTIL A>10
80 !
```

This is an indefinite repeat. The loop of lines 50, 60, and 70 keeps going until the variable A is greater than 10. Then execution exits the loop and continues with line 80. If A never becomes greater than 10, the loop continues until you press [RESET] or turn off the computer.

The REPEAT-UNTIL loop is a good way to check whether a value is correct or within a range.

Example: Young clerk William Guppy is fond of games of chance. He's written a program that lets you guess a number in the range from one to 10. To see and try the program, load GUESS_GAME from the disk of examples, or type it in using the listing below:

```
LOAD "GUESS_GAME" _
LIST _
```

```
10 ! RE-STORE "GUESS_GAME"
20 I=0
30 RANDOMIZE
40 A=INT(RND*10)+1
50 REPEAT
60 INPUT "Enter your guess, 1-10",N
70 I=I+1
80 UNTIL A=N
90 PRINT "Congratulations, that took only";I;"guesses!"
100 END
```

Line 30 of the program generates a random seed for the random number generator in line 40. The expression $\text{INT}(\text{RND} * 10) + 1$ in line 40 "scales" the number so it's in the proper range of 1 through 10.

Once the random number A is generated, the program begins executing the REPEAT-UNTIL loop. The loop waits for you to input a number N in line 60, then repeats. The loop continues until your guess (N) is finally the same as the number generated (A).

To keep track of how many guesses it takes, Guppy has included a loop counter I. Each time through the loop, 1 is added to the counter. When you finally guess correctly, I tells the world how many guesses you needed. Give it a try!

RUN_↓

Enter your guess, 1-10

3_↓

Enter your guess, 1-10

5_↓

Enter your guess, 1-10

9_↓

Congratulations, that took only 3 guesses!

The REPEAT-UNTIL loop is endless, if you don't guess right. But if you want to weasel out of the loop without giving the correct answer, you can press [STOP], [PAUSE], or [RESET].

Indenting to Taste

As you've worked through this course, you've probably seen that HP BASIC automatically indents some parts of your program (such as the statements within a loop) and outdents others (such as subroutine labels) to make it easier to read. But you can set indenting just the way you like it – with the `INDENT` command.

With the `GUESS_GAME` program still loaded, type this:

```
INDENT 5,3_↵  
EDIT_↵
```

You can see what happened to the way the program appears:

```
10      ! RE-STORE "GUESS_GAME"  
20      I=0  
30      RANDOMIZE  
40      A=INT(RND*10)+1  
50      REPEAT  
60          INPUT "Enter your guess, 1-10",N  
70          I=I+1  
80      UNTIL A=N  
90      PRINT "Congratulations, that took only";I;"guesses!"  
100     END
```

The command `INDENT 5,3` moved the starting column for showing the first character of most statements to column 5. It also specified an indent of three positions for "nested" structures – things like loops. `INDENT` makes it a lot easier to read a program listing; it lets you see at a glance just where the beginning and end of loops are, and where labels are.

Notice that `INDENT` doesn't move remarks that begin with an exclamation point. (It does move `REM` remarks, though.)

To remove all indenting, use 0 for the second number. Try it:

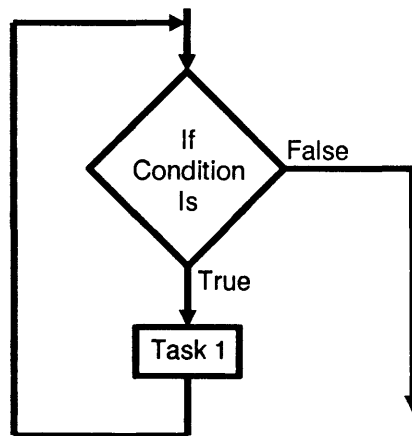
```
INDENT 5,0
```

All indenting is removed:

```
10      ! RE-STORE "GUESS_GAME"  
20      I=0  
30      RANDOMIZE  
40      A=INT(RND*10)+1  
50      REPEAT  
60      INPUT "Enter your guess, 1-10",N  
70      I=I+1  
80      UNTIL A=N  
90      PRINT "Congratulations, that took only";I;"guesses!"  
100     END
```

WHILE-END

With REPEAT-UNTIL, the loop is executed *until* a condition is true. In a WHILE loop, the instructions are executed over and over *while* a condition is true. When the condition becomes false, the loop is exited.



```
50 WHILE A<=10
60 GOSUB Task1
70 END WHILE
80 !
```

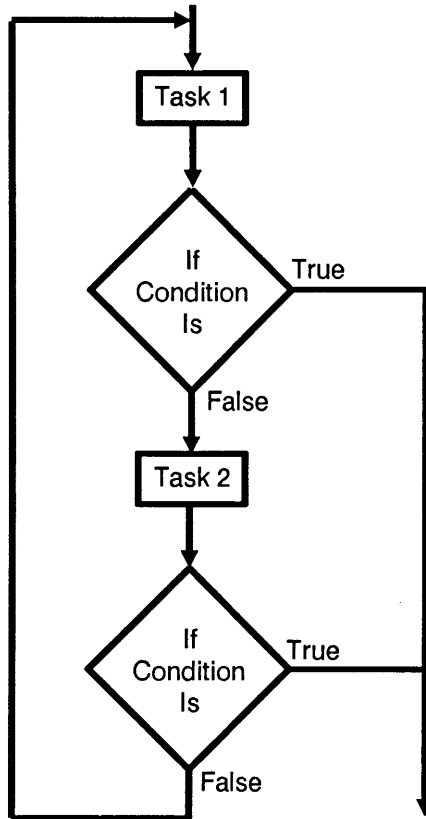
Here the loop is executed over and over as long as the value of A is less than 10. As soon as A is equal to or greater than 10, execution exits the loop and continues with line 80.

LOOP-END

The most powerful loop uses a statement called LOOP. The loop is between LOOP and END LOOP, with exits from the loop if an EXIT IF condition is true.

```
40 !
50 LOOP
60 GOSUB Task1
70 EXIT IF A=2
80 !
90 GOSUB Task2
100 EXIT IF B=3
110 END LOOP
120 !
```

In this example, the loop is between lines 50 and 110. The loop is endless unless A = 2 or B = 3. If either of these conditions becomes true, execution exits the loop and continues with line 120.



You can have any number of conditions, each with its own EXIT IF statement. By using LOOP and END LOOP with EXIT IF statements, you can provide multiple ways of exiting the loop while changing tasks and conditions within the loop.

Live Keyboard

One feature of HP BASIC that's especially useful during loops is the active, or "live," keyboard. You can read the value of a variable, or change its value, while a program is running.

Enter the simple program shown below:

```
SCRATCH,↵  
EDIT,↵
```

```
10 PRINTER IS 1  
20 WHILE I<100  
30 I=I+1  
40 PRINT I  
50 WAIT .1  
60 END WHILE  
70 END
```

Now press RUN. The program should begin displaying the current value of I:

```
RUN,↵
```

```
1  
2  
3
```

Now put your fingers on the keyboard and type:

```
I,↵
```

Look at the display line. You see the value to which I was set when you executed I=1. Even though the program continues printing new values for I, you still see the old one on the display line:

```
3
```

This shows an important fact about HP BASIC – the keyboard is "live" even during program execution.

You can use the live keyboard to change a variable's value in mid-program, too. With the same mini-program running, put your fingers on the keyboard and type:

```
I=1
```

See what happened? The program uses your new value for I and begins the loop again with that new value. Try it again:

```
RUN
```

```
1  
2  
. .  
45  
46
```

```
I=1
```

```
1  
2
```

The live keyboard can change the number assigned to a variable just as if a LET statement was executed in a program.

Functions

Remember what a *function* is? If you worked through lesson 1, you know that a function usually takes an argument and gives you a result. HP BASIC already has some functions in it, such as:

PI	Returns the value of π .
SQRT (45)	Returns the square root of 45.

You can also write your own functions for use in programs.

A function is like a subprogram. (You learned all about those earlier in this lesson.) Like a subprogram, a function:

- Is located after a program's END statement.
- Has local variables that can have different values—or even different meanings—in another part of the program.
- May require that you reserve memory and declare an OPTION BASE within the function. (You'll learn more about OPTION BASE in lesson 8.)

Let's create a program with a simple function that converts temperature in Fahrenheit to Celsius (older thermometers call it Centigrade), using the formula $C = 5/9(F - 32)$.

SCRATCH_┘

EDIT_┘

```
10 INPUT "Enter degrees F",F
20 PRINT F;"degrees F is";FNCE1(F);"degrees C"
30 END
40 !
50 DEF FNCE1(F)
60 C=5/9*(F-32)
70 RETURN INT(C)
80 FNEND
```

Now run the program to find the Celsius equivalent of 40 degrees Fahrenheit:

RUN_

Enter degrees F

40_

40 degrees F is 4 degrees C

How it works: The function itself is in lines 50 through 80. It is named FNCel. (All function names begin with FN.) The parts of the function are:

- DEF FN statement: Line 50 defines the function as FNCel(F).
- RETURN statement: The RETURN statement in line 70 is different from a RETURN in a subroutine. Here it means "return a value from this function to the calling program."
- FNEND statement: Line 80 ends the function.

The main program uses a function just as if it were one of the functions resident in HP BASIC. In line 20, the function operates on the current value of variable F, and replaces the expression FNCel(F) with a calculated value.

Since FNCel is now a function, you can use it from the keyboard just as you do any function.

For example, if it's 100 degrees Fahrenheit, how hot is it in degrees Celsius? Type:

FNCel(100)_

The answer is a steamy 37 degrees C:

37

Local Variables

Like those of a subprogram, a function's variables are all *local* – they have no meaning to other parts of the program. So even though line 30 of the program calls the function as FNCel(F), you don't have to use F as the variable within the function (in lines 50-80).

Try this: change lines 50 and 60 so the function uses – oh, say, Heat as a variable.

EDIT 50_↓

```
50 DEF FNCel (Heat)
60 C=5/9*(Heat-32)
70 RETURN INT(C)
80 FEND
```

Run the program now to find the Celsius equivalent of 0 degrees F:

RUN_↓

Enter degrees F

0_↓

0 degrees F is -18 degrees C

Functions vs. Subprograms

Later in this course you're going to find that functions and subprograms can pass parameters to the main program and back. For now, though, just remember:

- If you want to take data and generate a single value, use a function.
- If you want to manipulate data, do input/output, or generate more than one value, use a subprogram.

Review Quiz

1. You're running a program for the first time, and it seems to be in an endless loop. (It prints "What if?" over and over again.) How can you stop the program and exit the loop without turning off the computer?
2. Modify the program GUESS_GAME from the disk of examples to EXIT IF Guppy types 0. Use LOOP and END LOOP with an EXIT IF statement in between.
Modify line 60 to read "Enter your guess, 1-10 (0 to exit)." If Guppy exits by pressing 0, print "You failed to guess right after ___ guesses."
3. As years pass, the data paths of popular microprocessors increase by powers of two: first 16 bits (2^4), then 32 bits (2^5) and so on.
Write a program that predicts the future of the microprocessor; that is, that prints all the powers of 2 up to and including 2^{101} . Use a REPEAT-UNTIL loop and exit when you reach 1.27×10^{30} . (Careful with that comparison!)
4. Write the program LAUNCHER. It should count down from 100 to 0, displaying each number. When 0 is reached, the program should display the words "BLAST-OFF!"
5. Write a function FNR that rounds any number to two decimal places.

Laboratory Exercise

This exercise tests what you've learned in lessons 6 and 7.

Many programmers overuse the program statement **GOTO**. HP BASIC has a number of other statements that are much more efficient and easy to use. **GOSUB**, **IF-THEN-ELSE-END IF**, **SELECT CASE**, **FOR-NEXT STEP**, **REPEAT-UNTIL**, **WHILE-END WHILE** are all better than **GOTO**.

Here's what to do:

1. Load **LAB7_SHL** from your disk of examples. The program works, but it's convoluted and difficult to follow. You may want to run it once to see how it works. It counts down from 10 to 0, then back up from 0 to 10. After that, it asks you to input a value for voltage from the keyboard. Depending on the value you type, you'll see a different message.
2. List the program to see the code.
3. Before you change a line of code, plan how you're going to proceed. Then change lines according to the instructions in the program's remarks.

Hint: The main part of your program should look like the code on the next page.


```
Main!  
GOSUB Count_down  
GOSUB Count_up  
GOSUB Process  
DISP "Program terminated"  
STOP
```

There's a solution on your disk of examples; you'll find it as SOL_LAB7.

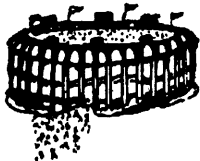
The Marvelous Array

Congratulations! You're now an expert on the use of variables and loops. With those under your belt, you're ready to learn about giant arrays, made up of individual numbers or strings.

In this lesson you'll learn about:

- **What an array is.**
- **Subscripted variables.**
- **Dimensioning with DIM.**
- **Setting the OPTION BASE.**
- **Putting data into an array.**
- **String arrays.**
- **Using RANK, SIZE, BASE, and SUM to investigate unknown arrays.**
- **Structured programming.**
- **Sorting.**

What Is an Array?



An array is a "holding area" in the computer's memory. It holds data in the form of numbers or strings or both. You use arrays to keep and manipulate the large amounts of data you need.

Each number in the array is in a unique location—it's like a sports stadium, where each spectator's ticket places him or her in a specific seat. A ticket reading A-5-M, for instance, would plop you down in section A, the 5th row, seat M.

Arrays can have many dimensions. An array with one dimension is a single row of data, like this:

1	2	3	4	5	6
23	4	9	85	27	2

You can think of a two-dimensional array as having both "rows" and "columns:"

		Columns					
		1	2	3	4	5	6
Rows	0	23	4	9	85	27	2
	0	3	9	18	4	0	17

HP BASIC allows arrays of up to six dimensions.

Subscripted Variables

Each piece of information in an array is assigned to a variable. These are called *subscripted variables*; they're shown with parentheses after them, like this:

```
Array (5)
C(4,2)
Seat (2,12,3,35)
```

The name of the entire array is shown by the variable itself, so Array, C, and Seat are all names of arrays.

The numbers in parentheses (the *subscripts*) show the location of a particular piece of data within the array. The variable C(4,2) shows the location of one piece of data within the array named C. The variable C(4,3) shows the location of the adjoining piece of data.

You use subscripted variables just as you do ordinary simple variables:

```
10 Array(5)=45
110 C(4,2)=C(4,2)*PI
80 PRINT "The winner is"; Seat
(2,12,3,35)
```

Putting Data into an Array

There are three steps to putting data into an array:

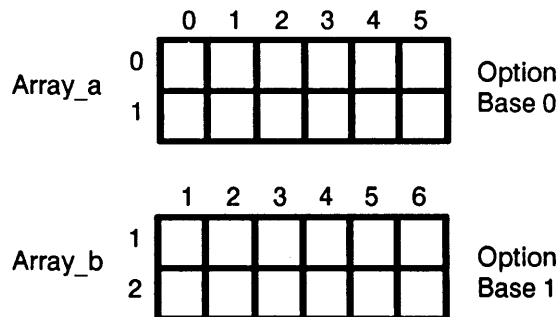
1. *Set the option base:* Decide whether the lowest element of the array will be at 1 or 0 (that is, have a subscript of 1 or 0).
2. *Dimension the array:* Decide the size and subscripts of the array and reserve memory space for it.
3. *Fill the array:* Fill the array with data.

Setting the Base Element

Your first question to answer is "what is the lowest element of the array?" You make this decision using the OPTION BASE statement.

OPTION BASE 0 sets the subscript of the base element to zero. OPTION BASE 1 sets the base element to one.

Look at these two arrays:



The two arrays are the same size. The subscript of the lowest element of Array_a, though, is 0. The lowest element of Array_b has a subscript of 1.

The first row of Array_a, then, consists of subscripted variables Array_a(0,0), then Array_a(0,1) then Array_a(0,2) and so on. The second row comprises Array_a(1,0), Array_a(1,1), Array_a(1,2), etc.

Since you've set Array_b to OPTION BASE 1, the subscript of its lowest element is 1. The subscripted variables begin with Array_b(1,1), Array_b(1,2), and so on. The highest-numbered element of Array_b is Array_b(2,6).

You can use OPTION BASE only once in any program or subprogram. You set the OPTION BASE for *all* arrays at once.

The computer "wakes up" set to OPTION BASE 0, so if you forget to use an OPTION BASE statement in your program, all arrays are automatically set with their lowest elements at zero. To prevent confusion, it's better to always declare the base element early in your program. Use something like:

```
10 OPTION BASE 0
```

or

```
10 OPTION BASE 1
```

Dimensioning the Array

Your next step is to set the dimensions of the array – that is, its size and shape.

The DIM statement is one way of dimensioning an array. Look at this line:

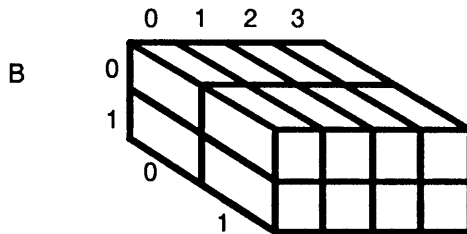
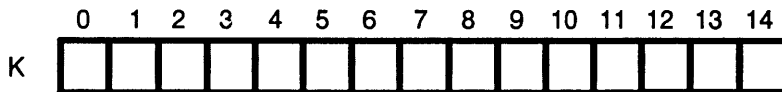
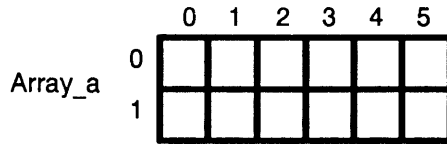
```
20 DIM M(2, 10)
```

Line 20 reserves space in memory for a two-dimensional array of 20 elements. The array is named "M", and its structure is 2 rows by 10 columns.

One DIM statement can dimension more than one array. For example:

```
10 OPTION BASE 0
20 DIM Array_a(2,6), K(15), B(2,2,4)
```

Line 20 creates three arrays. They look like this:



You can, if you like, use more than one DIM statement in a program. DIM dimensions and reserves memory for real numeric arrays, strings, and string arrays.

You can use DIM to specify the actual bounds of the array, too. Just use a statement with colons, like this:

```
DIM B(1:5, 2:6)
```

This dimensions an array of five rows and five columns. The rows begin with subscripts of 1 through 5, the columns with subscripts of 2 through 6.

	2	3	4	5	6
1					
2					
3					
4					
5					

DIM isn't the only statement for reserving memory, either. Look at these examples of other statements that dimension arrays:

```
10 INTEGER A(2,10)
```

Line 10 dimensions array A as a 2 x 10 array of integer numbers.

```
50 REAL Array_b(15)
```

Here, line 50 dimensions an array with one row of 15 full-precision real numbers.


```
100 COMPLEX A(512)
```

The statement in line 100 declares a one-dimensional array of 512 complex variables and reserves memory space for them.

```
40 COM INTEGER Array (-128:127)
```

Line 40 declares an array of integer values in a common area (for use by more than one variable). The elements are from Array(-128) to Array(127).

```
4180 ALLOCATE Temp$(N)[100]
```

Line 4180 dynamically allocates memory for an array of N strings of 100 characters each.

DIM and COM must be placed after OPTION BASE (if you use an OPTION BASE statement), and before you refer to the variable or array.

A program can have more than one DIM, COM, or ALLOCATE statement. You can reserve memory for a particular array or variable only once using DIM and COM.

ALLOCATE lets you reserve memory dynamically – that is, when you don't know how large an array will be. Unlike memory reserved with DIM, ALLOCATE makes a temporary reservation. You can later DEALLOCATE that memory and use it for something else.

Putting Data into the Array

Remember, an array is just a group of variables, so when you "put data into the array," you're just setting those variables equal to numbers or strings. You can put data into an array using statements you've already used for this purpose: LET, INPUT, and READ, for example.

Suppose you wanted an array of the squares of whole numbers. You could fill the array with data using individual LET statements (or implied LET statements since you don't need the LET keyword).

```
30 LET A(1)=1
40 A(2)=4
50 A(3)=9
60 LET A(4)=16
70 LET A(5)=25
```

Lines 30-70 show one way to fill an array—the hard way.

Using a Loop

The beauty of an array is that it makes handling large amounts of data easy. Try this mini-program to see a fast, efficient way to fill array A:

```
SCRATCH_┘
EDIT_┘
```

```
10 OPTION BASE 1
20 DIM A(5)
30 FOR I=1 TO 5
40 A(I)=I^2
50 NEXT I
60 PRINT A(*)
70 END
```

Now run the program:

RUN_

```
1   4   9  16  25
```

See how it works? In line 10, you declared an **OPTION BASE** of 1; in line 20 you dimensioned an array **A**, consisting of five elements. So the elements are **A(1)**, **A(2)**, **A(3)**, etc.

The **FOR-NEXT** loop in lines 30-50 fills the array. The first time through the loop, $I = 1$, so **A(I)** is **A(1)**. The statement $A(I) = I^2$ sets this element equal to 1 squared.

The second time through the loop, $I = 2$, so line 40 sets $A(2) = 2 \times 2$.

When the **FOR-NEXT** loop is completed, the **PRINT(*)** statement in line 60 prints the values in the array.

Using the Asterisk

In line 60, the asterisk means "all elements of the array." If you put a semicolon after the asterisk, it prints the elements close together. To try it:

```
EDIT 60_↵  
60 PRINT A(*) ;_↵  
RUN_↵
```

The elements of the array are printed close together:

1 4 9 16 25

Using Elements of an Array

Once an array is filled, you can use its elements just as you do any variable. For instance, you can use the keyboard to see the value of any subscripted variable in the array:

```
A(3)_↵
```

9

```
A(5)_↵
```

25

Strings in Arrays

Naturally, arrays aren't limited only to numbers. You can also use strings of characters and words in arrays.

Example: For Jonas Chuzzlewit, today's friends are tomorrow's enemies, so he's written a program, FRIENDS, that keeps track of who his friends are. Load it now and run it:

```
LOAD "FRIENDS" _  
RUN _
```

Type a friend's name

Start typing the names of friends. Answer the question "Any more names?" with Yes until you can't think of any more. Then answer the question with No.

```
Mark Tapley _
```

Any more names? (Yes/No)

```
Yes _
```

Type a friend's name

```
Charity Pecksniff _
```

Any more names? (Yes/No)

```
Y _
```

Type a friend's name

Sarah Gamp↵

Any more names? (Yes/No)

No

Your real friends are:
Mark Tapley Charity Pecksniff Sarah
Gamp
You have 3 friends

How it works: To see how the FRIENDS program works,
list it:

LIST↵

```
10 ! RE-STORE "FRIENDS"
20 OPTION BASE 1
30 DIM Friend$(100)[40]
40 I=0
50 CLEAR SCREEN
60 REPEAT
70 I=I+1
80 INPUT "Type a friend's name",Friend$(I)
90 INPUT "Any more names? (Yes/No)",No$
100 No$=UPC$(No$[1,1])
110 UNTIL No$="N"
120 PRINT "Your real friends are:"
130 PRINT Friend$(*)
140 PRINT "You have";I;"friends"
150 END
```

Line 30 reserves space for a string array called Friend\$; the array has 100 elements, and each element has space for a 40-character name.

A REPEAT-UNTIL loop in lines 60-110 lets you type names into the array. Each time through the loop, the loop counter I is incremented. The INPUT statement in line 80 stops to let you type in a name; that name is assigned to the current variable Friend\$(I).

The first time through the loop, this happened:

```
I=1  
Name$(1)="Mark Tapley"
```

The second time through the loop, I was incremented, so:

```
I=2  
Name$(2)="Charity Pecksniff"
```

Line 90 asks you to answer the question with "Yes" or "No", then waits for your input. The statement No\$ = UPC\$(No\$[1,1]) in line 100 sets the variable to the uppercase first character of the answer you type. If you type anything except a word beginning with N, the loop is repeated.

The loop repeats UNTIL you answer the question with a word beginning with N (such as "No", "no", "NONE", "NEVER"). Then the variable No\$ is equal to "N"; this causes execution to exit the loop.

The PRINT Friend\$(*) statement in line 130 means "print all elements of the array named Friend\$."

The Bubble Sort

Among the ways that computers can manipulate numbers and strings in arrays, nothing is handier than *sorting*. You can sort names into alphabetical order, or numbers into ascending or descending order.

The "bubble sort" (so-called because the lowest number or word "bubbles" to the top) is one way of sorting data. In a bubble sort, two nested loops are used, and a temporary variable allows switching the positions of two elements of the array.

This program features a bubble sort of three names. (You'll find the program as BUBBLE on your disk of examples.) By changing the dimensioning and the value of N, you can use it to sort any number of elements.

```
10 ! RE-STORE "BUBBLE" Sort
20 OPTION BASE 1
30 N=3
40 DIM Name$(3)[20], Temp$[20]
50 DATA Noggs, Pipchen, Claypole
60 READ Name$(*)
70 FOR I=1 TO N-1
80   FOR J=I+1 TO N
90     IF Name$(I)>Name$(J) THEN GOSUB Exchange
100    NEXT J
110 NEXT I
120 PRINT Name$(*)
130 STOP
```




```

140 Exchange: !
150 Temp$=Name$(I)
160 Name$(I)=Name$(J)
170 Name$(J)=Temp$
180 RETURN
190 END

```

Let's "walk through" the sorting process.

After the READ statement, the three names are in the array like this:

Variable	Name
Name\$(1)	Noggs
Name\$(2)	Pipchen
Name\$(3)	Claypole

1. The first time through the J loop, the counters are set as shown here:

I	Name\$(I)
1	Noggs
J	Name\$(J)
2	Pipchen

In the alphabet, N is lower ("less than") P, so line 90 is false: Noggs is not greater than Pipchen. The positions of the names aren't changed.

2. Look what happens the next time through the J loop: The variable I stays the same (1), but the variable J is incremented to 3:

I	Name\$(I)
1	Noggs
J	Name\$(J)
3	Claypole

Now line 90 is true: Noggs is greater than Claypole in the alphabet. The IF-THEN statement causes a branch to the Exchange subroutine in line 140.

Within the Exchange routine, the positions of Noggs and Claypole are swapped. It goes like this:

1. Line 150 copies Noggs into Temp\$. (Remember, what's on the right of the equals sign goes into the variable on the left.)

```
150 Temp$=Name$( I)
```

Name\$(I) Noggs	Name\$(J) Claypole	Temp\$ Noggs
---------------------------	------------------------------	------------------------

2. Line 160 copies Claypole into Name\$(I).

```
160 Name$( I)=Name$( J)
```

Name\$(I) Claypole	Name\$(J) Claypole	Temp\$ Noggs
------------------------------	------------------------------	------------------------

3. Finally line 170 copies Noggs into Name\$(J).

```
170 Name$( J)=Temp$
```

Name\$(I) Claypole	Name\$(J) Noggs	Temp\$ Noggs
------------------------------	---------------------------	------------------------

You can see that Claypole and Noggs have switched places, so the array now looks like this:

Variable	Name
Name\$(1)	Claypole
Name\$(2)	Noggs
Name\$(3)	Pipchen

You can sort huge arrays of names (or numbers) with virtually the same program. Just increase the value for the number of elements, N.

Special Array Functions



A number of special functions in HP BASIC let you identify the elements of an array, copy arrays into other arrays, add all the elements of an array, search an array, and do other useful tasks. Let's try a few of them on another array.

Example: The proprietor of Nadgett's Investigations is in the midst of his latest caper, delving into the mysteries of a particular HP BASIC array. He has one clue, and only one. And though he won't admit it, he needs you – and your knowledge of arrays – desperately.

To see Nadgett's single clue, load and run the program MAGIC from the disk of examples. But be fair – don't list it yet.

```
LOAD "MAGIC" ↵  
RUN ↵
```

I'm the mystery array. My name is M.
What else can you find out about me?

Finding the Dimensions

You know the array's name is M. You can use that name to find out other things about the array. For instance, to find out how many dimensions it has, use the RANK function. Type:

```
PRINT RANK (M) ↵
```

The display line shows the answer:

2

RANK returns the number of dimensions of an array. In the case of **M**, you now know it has two dimensions: rows and columns.

Armed with this knowledge, you can now determine the array's size. For this you'll need the **SIZE** function. **SIZE** returns the number of elements in a dimension of an array. Try this:

```
PRINT SIZE (M,1)↵
```

5

The **SIZE** function here means "how many elements in **M**'s first dimension?" The answer is 5, so you know it has five rows.

But **M** has two dimensions: use **SIZE** again to find out how many elements in the second dimension:

```
PRINT SIZE (M,2)↵
```

5

Aha! You're dealing with a 5 x 5 array.

Finding Out the Option Base

Now what about the variables themselves. How are they addressed? Use the **BASE** function to determine the lower bounds of the variable subscripts in the rows and columns:

```
PRINT BASE (M,1)↵
```

1

```
PRINT BASE (M,2)↵
```

1

Now you're getting somewhere! This mystery array is set to **OPTION BASE 1**. The elements are $M(1,1)$, $M(1,2)$, $M(1,3)$ and so on, right up to $M(5,3)$, $M(5,4)$, and $M(5,5)$.

	2	3	4	5	6
1					
2					
3					
4					
5					

Finding Out the Data

What about the data that's in the array? Nadgett needs to know that, too.

Remember the mighty asterisk? Use it with **PRINT** to see a list of the data in the array:

```
PRINT M(*);
```

```
14 10 1 22 18 20 11 7 3 24 21 17 13 9 5 2  
23 19 15 6 8 4 25 16 12
```

That's all the numbers in the array. They're printed in a long line, but since you've uncovered information that they're in the array in five columns and five rows, even Nadgett should be able to determine exactly what the array looks like.

Summing the Array

One other thing you can do for Nadgett is tell him the sum of all the elements. Use the SUM function, like this:

```
DISP SUM(M)↵
```

```
325
```

The sum of all those numbers is 325.

At this point, Nadgett is off writing a mini-program to print the array. However, you can do the same thing by merely modifying one line of the program. Type:

```
EDIT 50↵
```

```
50 !GOSUB Print
```

There's the culprit! The exclamation point here "comments out" the GOSUB call, so the array is never printed. Remove the exclamation point (don't forget to "enter" the line by pressing ENTER↵) Then run the MAGIC program again:

```
50 GOSUB Print↵
```

```
RUN↵
```

```
14 10 1 22 18
20 11 7 3 24
21 17 13 9 5
2 23 19 15 6
8 4 25 16 12
```

There's the array, all right! If you look closely, you'll see why Nadgett pursued his investigation so intently: this array is a "magic square." If you add up the numbers in any row, any column, or even any diagonal, you'll find the same sum. Nadgett knows that ancient seers ascribed great and mystical properties to magic squares – and now, thanks to you, he has one of his very own!

Here's a listing of MAGIC:

```
10 ! RE-STORE "MAGIC"
20 GOSUB Init
30 GOSUB Data
40 GOSUB Clue
50 !GOSUB Print
60 STOP
70 Init: !
80 CLEAR SCREEN
90 OPTION BASE 1
100 DIM M(5,5)
110 RETURN
120 Data: !
130 READ M(*)
140 DATA 14, 10, 1, 22, 18, 20, 11, 7, 3, 24, 21, 17, 13, 9, 5, 2, 23, 19, 15, 6, 8, 4, 25,
16, 12
150 RETURN
160 Clue: !
170 PRINT "I'm the mystery array. My name is M."
180 PRINT "What else can you find out about me?"
190 RETURN
200 Print: !
210 PRINT M(1,1), M(1,2), M(1,3), M(1,4), M(1,5)
220 PRINT M(2,1), M(2,2), M(2,3), M(2,4), M(2,5)
230 PRINT M(3,1), M(3,2), M(3,3), M(3,4), M(3,5)
240 PRINT M(4,1), M(4,2), M(4,3), M(4,4), M(4,5)
250 PRINT M(5,1), M(5,2), M(5,3), M(5,4), M(5,5)
260 RETURN
270 END
```

Structured Programming

Take a good look at the structure of MAGIC. It's a good example of a technique known as a "structured" program.

The actual program – the main part – is separated into a number of different tasks, each performed by its own subroutine. The main part of the program (lines 20-60) is very short and consists entirely of subroutine calls.

This structure makes it easy to "comment out" different parts of the program when you're writing it. That way, you can run and debug one section of code at a time. Then, when all the individual sections run correctly, you strip off the comments and voila! Your entire program is bug-free.

Review Quiz

1. Make a quick drawing of the array Amps using the information in these two program lines:

```
10 OPTION BASE 0
20 DIM Amps (3,5)
```

2. What's wrong with this program?

```
10 PRINTER IS 26
20 CLEAR SCREEN
30 OPTION BASE 1
40 FOR C= 1 TO 20
50 A(C)=C*2
60 NEXT C
70 FOR P= 1 TO 19
80 PRINT P;" ";A(P+1)*A(P)
90 NEXT P
100 END
```

3. This program sorts numbers according to size. Will the smallest or largest number appear at the top of the list?

```
10 OPTION BASE 1
20 DIM A(20)
30 FOR C= 1 TO 20
40 DISP
50 DISP "Please enter any number"
60 INPUT A(C)
70 NEXT C
80 FOR K= 1 TO 19
90 FOR L= 1 TO 19
100 IF A(L)>A(L+1) THEN 140
110 T=A(L)
120 A(L)=A(L+1)
130 A(L+1)=T
140 NEXT L
150 NEXT K
160 FOR C= 1 TO 20
170 PRINT A(C)
180 NEXT C
190 END
```


Printing to Please

It's sad but true – no matter how powerful your programs or how elegant your solutions to knotty programming problems, nobody looks at your actual code. People see only the *output* that's printed or displayed.

This lesson shows how to turn the output from your programs into a thing of beauty. You'll learn about:

- Printing in fields on your screen and printer.
- Formatting output with PRINT USING.
- The IMAGE statement.
- Multiple images in one statement.

Formatting the Easy Way

When you control the appearance of a number or string to print it out, it's called *formatting*.

Formatted printing changes only the appearance of the item – it doesn't affect the actual number or string that's in the variable. No matter how a number looks, you can be assured that calculations using any real number are always performed with the full accuracy of that number.

You already know one way of formatting your printed output. You learned it clear back in lesson 3. Do you remember what it is?

Here's a hint. Type and run this mini-program:

SCRATCH ↵

EDIT ↵

```
10 A$="HELLO"  
20 B=130295  
30 C$="WELCOME TO SIRIUS"  
40 PRINT A$,B,C$  
50 PRINT A$;B;C$  
60 END
```

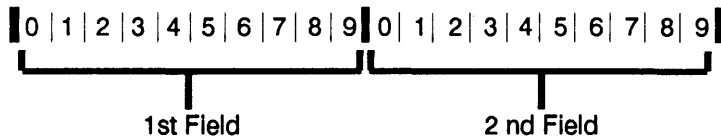
Be sure you separate the items in line 40 with commas (,). Use semicolons (;) in line 50. Then run the program:

RUN ↵

```
HELLO    130295    WELCOME TO SIRIUS  
HELLO 130295 WELCOME TO SIRIUS
```

You remember now, don't you? The semicolon causes items to be printed side by side (as in line 50), while a comma puts space between them.

How it works: What actually happens is that the display screen or print area is divided into *fields*. There is room for 10 characters in each field.



When you use a comma between items, the first item is printed in the left-most field. The next item begins printing at the beginning of the next field.

```
PRINT A$,B,C$_
```

```
HELLO 130295 WELCOME TO SIRIUS
```

↑ ↑
New fields begin here

When you use semicolons, the fields are ignored, and the items are printed close together. As you know, for a number that means one space (or a sign) before, and one space after:

```
PRINT A$;B;C$_
```

```
HELLO 130295 WELCOME TO SIRIUS
```

When you use semicolons to separate strings, they're printed with no spaces between them:

```
PRINT A$;C$_
```

```
HELLOWELCOME TO SIRIUS
```

More Sophisticated Formatting

Using commas and semicolons is the easiest way to control what your printed output looks like. For more control and sophistication, try a PRINT USING statement.

For an example of what PRINT USING can do, type and run this mini-program:

```
SCRATCH_↵  
EDIT_↵
```

```
10 FOR I=1 TO 3  
20 PRINT RND  
30 NEXT I  
40 END
```

```
RUN_↵
```

```
.0174532923929  
.337485246983  
.114546045249
```

The program prints three random numbers. (The RND function in line 20 generates a random number from a "seed." Since we haven't told the computer to RANDOMIZE first, the number's seed is always the same. This makes the "random" numbers in this example always the same, too.)

As it is now, the program prints a *lot* of digits for each number. Let's put a PRINT USING statement in line 20 to make these numbers easier to read. Change line 20 exactly as shown.

```
EDIT 20_␣  
20 PRINT USING "S2D.3D";RND_␣
```

Then run the program again:

```
RUN_␣
```

```
+ .017  
+ .337  
+ .115
```

Isn't that better? The numbers are rounded off to three decimal places, and each has a sign (+) attached.

Here are some of the things you can do with PRINT USING:

- Eliminate leading and trailing blanks.
- Line up all decimal points.
- Make columns right-justified (that is, lined up along the right instead of the left).
- Insert commas in long numbers for readability.
Control the number of digits shown.

In short, PRINT USING makes your printed output cleaner, tidier, and easier to read.

Using PRINT USING

To format your printed output, put a PRINT USING statement in your program. Look at the statement in our example again:

```
20 PRINT USING "S2D.3D";RND
```


The characters "S2D.3D" form the *image* in which the number is printed. The parts of this particular image mean:

- The "S" adds a sign to what's printed.
- The "2D" prints two digits.
- The "." prints (can you guess?) the decimal point.
- Then the "3D" prints three more digits.

The semicolon (;) separates the image from what you want to print (RND, in this case).

Printing Numbers

Besides "D" and the decimal point, there are several other specifiers you can place in your PRINT USING statements. Some you'll use with numbers, some with strings, and some with all types of output. You'll find a complete list back in lesson 25.

Here are the most common specifiers you'll use in images to print numbers:

Image Specifier	What It Does
D	Prints a digit. If this is a leading zero, a blank or the number's sign is printed.
Z	Same as D, except leading zeros are printed.
.	Prints the decimal point.
S	Prints the sign of the number: either + or -.
M	Prints - sign if the number is negative; print a blank if number is positive.
E	Prints "E" followed by a sign and two digits of the exponent. This is the same as "ESZZ".

Image Specifier	What It Does
ESZZ	Prints "E" followed by a sign and two digits of the exponent.
ESZZZ	Prints "E" followed by a sign and three exponent digits.
K	Prints the entire number without leading or trailing spaces.
X	Prints a blank.
*	Like Z, except asterisks are printed instead of leading zeros.
5() (or any number)	Repeat the specifier or group of specifiers 5 (or any number of) times.

Let's print the number 123.4567 several different ways with PRINT USING.

PRINT USING "DDD.DD"; 123.4567_

123.46

The computer prints one digit for each D in your PRINT USING image. There are three digits to the left of the decimal point and two digits to the right. The "2D" after the decimal point rounds the number off to two digits and prints them.

You get the same effect with this:

PRINT USING "3D.2D"; 123.4567_

123.46

As you can see, numbers can be used as "multipliers" for image items such as D and Z. So "3D" is the same as "DDD".

Now try these other examples with the same number,
123.4567:

PRINT USING "K"; 123.4567_

123.4567

"K" prints the entire number.

PRINT USING "S5Z.2D"; 123.4567_

+00123.46

Here the "S" prints the sign. "5Z" prints two leading zeros
and three digits (filling a total of five spaces).

PRINT USING "SD.3DE"; 6.023E+23_

+6.023E+23

PRINT USING "S3D.3DE"; 6.023E+23_

+602.300E+21

These examples show how a number with an exponent can
be easily formatted with PRINT USING.

Asterisks in your printed output can help you see fields and
spaces better. And PRINT USING lets you put them into
your printouts:

PRINT USING "5*.DDD"; 123.4567_

**123.457

In this example, the computer tries to print asterisks in the five spaces to the left of the decimal point. Three spaces are occupied by numbers, so you see them, along with two leading asterisks. The three D's round the number to three digits to right of the decimal point.

Multiple Images in One Statement

Several images can be placed in a single PRINT USING statement. Just separate the images with commas, like this:

```
PRINT USING "DD,SDD,DD"; 10, 20, 30_
```

```
10+2030
```

The first element in the image, DD, formats the first number. The second number is printed immediately after the first, using the image SDD (sign plus two digits). Then the third number is printed using the third image (DD).

Notice that the PRINT USING image *overrides* the commas separating the numbers 10, 20, and 30, so the fields are ignored.

You can, of course, print to different fields on the same line, but you must set it up with the image in the PRINT USING statement. Try this mini-program to see an example:

```
SCRATCH_
EDIT_
```

```
10 PRINT USING "5( ""1234567890 "" )"
20 PRINT
30 PRINT USING "5D.4D";1.1, 22.22, 333.333, 4444.4444, 55555.55555
40 PRINT USING "5*.4D";1.1, 22.22, 333.333, 4444.4444, 55555.55555
50 END
```

RUN_┘

```
12345678901234567890123456789012345678901234567890
```

```
    1.1000   22.2200  333.3330 4444.444455555.5556  
****1.1000***22.2200**333.3330*4444.444455555.5556
```

See what happened? Line 10 printed numbers to show the fields and the position of each character. (Line 10 actually prints a string five times; the double quotation marks are necessary to output this string in a `PRINT USING` statement. You'll soon learn more about string output.)

Line 20 prints a blank line, then line 30 prints the numbers. It prints the first number (1.1) in the left-most field, then the next number (22.22) in the next field, and so on. Line 40 prints the same numbers with leading asterisks so you can see the positions better.

There are two major points to remember about lines 30 and 40 in this mini-program:

- You can use a single image to print several numbers.
- Here, it is the image "5D.4D" or "5*.4D" that makes each number occupy a single 10-space field.

Printing Strings

The PRINT USING statement can also help you print words, names, and other strings. You can use the image specifiers shown here for strings:

Image Specifier	What It Does
A	Prints a single character of the string (or trailing blank if all characters have been printed).
K	Prints entire string without leading or trailing blanks. (Good for printing strings of unknown length.)
X	Prints a blank (space) in this position.
"Characters"	Prints the characters that are between the quotation marks. (Use double quotation marks if the characters are inside the PRINT USING's quotation marks.)
5() (or any number)	Repeats the specifier or group of specifiers 5 (or any number of) times.

Let's go through a couple of examples. Try these:

```
PRINT USING "15A"; "TOO LONG INDEED"
```

```
TOO LONG INDEED
```

```
PRINT USING "8A"; "TOO LONG INDEED"
```

```
TOO LONG
```

When you specified 15A in your image, all 15 characters of the string "TOO LONG INDEED" were printed. But when you left room for only eight characters (via the image "8A"), only the first part of the string was printed.

Now try some other examples of the PRINT USING statement. Remember the "page header" that let you see the fields earlier? Try this:

```
PRINT USING "K"; "12345678901234567890"↵
```

12345678901234567890

That's a start, but it's sure a lot of numbers to type. Use this instead:

```
PRINT USING "3( ""1234567890"" )"↵
```

123456789012345678901234567890

(Notice that you needed double quotation marks around the characters since they're actually inside the PRINT USING's image.)

Now you're ready to type in some names with PRINT USING:

```
PRINT USING "5X,10A,2X,13A"; "RUTH PINCH",  
"MONTAGUE TIGG"
```

123456789012345678901234567890 RUTH PINCH MONTAGUE TIGG
--

The PRINT USING statement printed five blank spaces (5X), then the 10 characters of Ruth Pinch's name (10A). Next it printed two more spaces (2X), and finally printed the 13 characters of Montague Tigg's moniker.

Other Uses for PRINT USING

Besides making your names and numbers look shipshape, a PRINT USING statement can do a lot of other work for you. Other image specifiers let this statement increase control over printing and your printer.

Look over this list of additional image specifiers for PRINT USING:

Image Specifier	What It Does
B	Specifies that one byte of data is output. The effect is to print the corresponding ASCII character (similar to CHR\$).
#	Suppresses current end-of-line (EOL) sequence. The effect is that the printer doesn't go to the next line after printing this line.
/	Sends a carriage return (CR) and line feed (LF). Use this to get the printer to advance the paper one line.
L	Sends the current end-of-line (EOL) sequence. On printers the EOL sequence is usually a CR/LF, so this is normally the same as the slash (/).
@	Sends a form feed to advance the paper one page.

There are other specifiers besides the ones shown here; you'll find a complete list in lesson 25. But these are the most common characters for specifying PRINT USING images.

Try this sequence in order:

```
PRINTER IS 1 ↵  
PRINT "PAGE 1" ↵  
PRINT USING "@,6A";"PAGE 2" ↵
```

```
PAGE 1
```

```
PAGE 2
```

After "PAGE 1" was printed, the PRINT USING statement sent a form feed to the current printer (the display screen). This advanced the "paper" one page, causing "PAGE 2" to be printed at the top of the next page.

Now try this mini-program:

```
SCRATCH ↵  
EDIT ↵
```

```
10 A$="SWEEDLEPIPE"  
20 B=77  
30 C=82  
40 PRINT USING "B,B,X,#";B,C  
50 PRINT A$  
60 END
```

```
RUN ↵
```

The output shows that you can get some interesting effects indeed with PRINT USING:

```
MR SWEEDLEPIPE
```

In this mini-program, the PRINT USING statement in line 40 prints variable B as a byte, prints variable C as a byte, and prints a space. The "#" suppresses the CR/LF that would usually cause line 50 to be printed on another line. So the output from line 40 and line 50 are printed together.

Using an Image

If you find yourself with the same PRINT USING statement over and over in a program, you can use it to refer to an IMAGE statement.

Here's an example:

```
100 Dataformat: IMAGE S3D.2D
```

This IMAGE statement has a label of "Dataformat". It says "print data with a sign, three digits, a decimal point, then two more digits." The colon (:) in line 100 isn't actually part of the label, but it identifies this statement as a label.

Notice that you don't need quotation marks in the IMAGE statement.

Now you can refer to the IMAGE statement each time you have PRINT USING:

```
100 Dataformat: IMAGE S3D.2D
110 PRINT USING Dataformat; A,B
120 PRINT USING 100; C,D,E
```

You can refer to the IMAGE statement by its label (Dataformat) or its line number (100).

Take a few minutes and try out this mini-program:

```
SCRATCH_↵  
EDIT_↵
```

```
10 Printform: IMAGE S5D.2D  
20 DATA 1.11,22.22,333.33,444.44,555.55  
30 READ A,B,C,D,E  
40 PRINT USING Printform;A,B,C  
50 PRINT USING 10;D,E  
60 END
```

The run the program to see the output:

```
RUN_↵
```

```
+1.11   +22.22   +333.33  
+444.44 +555.55
```

Notice that both PRINT USING statements use the same IMAGE, the one in line 10.

The IMAGE statement is one of the most powerful implements you have in your programming toolbox. You can refer to an IMAGE using all of these statements:

- PRINT USING.
- DISP USING.
- OUTPUT USING.
- ENTER USING.
- LABEL USING.

If you're doing instrument control, you'll use IMAGE not only for printing, but also for formatting instructions to instruments. Parts 2 and 3 of this course show how to use IMAGE with some of these other statements.

If you're doing instrument control, you'll use `IMAGE` not only for printing, but also for formatting instructions to instruments. Parts 2 and 3 of this course show how to use `IMAGE` with some of these other statements.

Review Quiz

1. What's another way to type the image "DDDDD.DDD"?
2. What does this code print?

```
10 Image1: IMAGE 32("*")
20 PRINT USING Image1
```

3. Look at this mini-program:

```
10 DATA 1234, ABCD
20 READ X, Y$
30 _____
40 END
```

Write a `PRINT USING` statement for line 30. It should print the data in line 10 as:

```
1234ABCD
```

4. See if you can predict the output from this mini-program:

```
10  DIM A$(62),B$(20)
20  A$="This shows how images give you data the way you like it."
30  B$="From Hewlett-Packard"
40  C=1234567
50  PRINT
60  PRINT USING "K,7D";"An example number is ",C
70  PRINT
80  PRINT USING """"Another example number is """,7D";C
90  Use_this: IMAGE 5X, 20A
100 PRINT USING "35A,/";A$
110 PRINT USING Use_this;B$
120 END
```

5. Remember the MAGIC program from lesson 8? Take a look at the listing again.

```
10 ! RE-STORE "MAGIC"
20 GOSUB Init
30 GOSUB Data
40 GOSUB Clue
50 !GOSUB Print
60 STOP
70 Init: !
80 CLEAR SCREEN
90 OPTION BASE 1
100 DIM M(5,5)
110 RETURN
120 Data: !
130 READ M(*)
140 DATA 14, 10, 1, 22, 18, 20, 11, 7, 3, 24, 21, 17, 13, 9, 5, 2, 23, 19, 15, 6, 8, 4, 25,
16, 12
150 RETURN
160 Clue: !
170 PRINT "I'm the mystery array. My name is M."
180 PRINT "What else can you find out about me?"
190 RETURN
200 Print: !
210 PRINT M(1,1), M(1,2), M(1,3), M(1,4), M(1,5)
220 PRINT M(2,1), M(2,2), M(2,3), M(2,4), M(2,5)
230 PRINT M(3,1), M(3,2), M(3,3), M(3,4), M(3,5)
240 PRINT M(4,1), M(4,2), M(4,3), M(4,4), M(4,5)
250 PRINT M(5,1), M(5,2), M(5,3), M(5,4), M(5,5)
260 RETURN
270 END
```

Look at the Print subroutine in lines 200-260. Rewrite these lines using two nested FOR-NEXT loops to print the array. Each row of numbers should be printed on a separate line, so you can see the magic square when you're done.

Laboratory Exercise

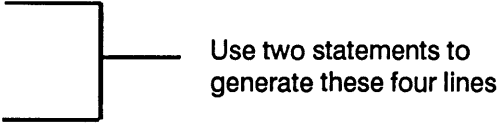
Load the program called LAB9_SHL from your disk of examples. This program is a shell – it's your job to fill in the necessary code.

In the program, you are given this data:

```
A=1234567
B=-1234567
C=.1234
D="FORMATTING"
```

Your new program lines should format and print the data on the CRT like this:

```
123456789012345678901234567890123456789012345678901234567890
FORMAT
-1234567.000
+1234567.0
  1234567
1.235E+006
-1234567.00
0.123      0.123      0.123
  1234567.00
  0000000.12
1.235E+06
```



Use two statements to generate these four lines

You should be able to generate the bottom four lines with just two statements.

The solution is also on your disk of examples, as SOL_LAB9.

Using Mass Storage

Throughout this course, you've learned about "computer memory." This is the memory inside your computer that's used by your programs. You can store words and numbers in memory (usually in arrays), but everything is wiped out – cleared completely – whenever you turn the computer off, lose power, or use SCRATCH.

Another kind of memory that you can use is called "mass storage." Mass storage can be the so-called "floppy" or "minifloppy" disks, hard disks, magnetic tape, and more.

In lesson 3 you learned how to initialize a disk and save programs on it. In this lesson you'll learn more details on using mass storage, and how to store data as well as programs.

You'll learn about:

- Using PRINT LABEL and READ LABEL to identify your disks.
- ASCII, BDAT, and HP-UX files.
- Using CREATE to create a data file.
- Using ASSIGN to open or close a path to a file.
- Serial and random access.
- Putting data in a file with OUTPUT.
- Reading data from a file with ENTER.

A Data Storage Example

Why put data into mass storage? That data may be priceless, for one thing. It may be the result of hundreds of experiments or thousands of hours of collection.

Or it may be voluminous—something like the New York City telephone directory—that your computer must process in small bites rather than in one large gulp.

The best way to learn about data storage is to "walk through" an example.

Example: In her current project, eminent engineer Emma Haredale uses the square roots of the numbers 1 through 100 over and over—so often, in fact, that she wants to store a "square root table" on disk, ready to load into an array for use. Can you help her?

How to do it: Whether you're storing programs or data, you'll need these two steps:

- Specify which mass storage unit.
- Initialize the disk in the storage unit (if it's a new disk).

Then, to store *data*, you'll also need to:

- Create a data file on the disk.
- Open a path to the data file.
- Output data along the path to the file.
- Close the path to the file.

Finally, to actually get and use the data again, you must:

- Open a path to the file.
- Enter the data from the file.
- Close the path.

You'll go through these steps one at a time.

Reviewing Program Storage

In lesson 3, you learned how to specify mass storage and initialize a disk. But before you put Emma Haredale's data on disk for her, take a moment to review *program* storage.

If you worked through lesson 3, you know how to specify mass storage and how to initialize a disk. You should also remember how to copy a program from your computer's memory to disk. Take a moment for a quick review.

Specifying Mass Storage

You use the MASS STORAGE IS or MSI statement to specify the disk or other mass storage unit, like this:

```
MSI ":CS80, 700, 0"
```

This statement makes the disk drive at interface 7, address 0, the default mass storage unit. It means that any STORE, SAVE, LOAD, or GET operations will automatically be to or from that disk.

Initializing a Disk

Unless a disk is new, you don't have to initialize it. In fact, before you initialize a disk, you should always find out if there is anything valuable on it:

```
CAT_┘
```

This catalogs the disk and checks to see what's on it. If the disk has already been initialized once, you don't have to do it again.

If you *do* want to initialize, use a statement like this:

```
INITIALIZE ":CS80,700,0" ↵
```

This statement initializes the disk – that is, it wipes out anything on the disk and prepares it to receive data or programs.

Storing a Program

To store a program as a PROG file, use the STORE or the RE-STORE statement with a program name, like this:

```
RE-STORE "FILENAME" ↵
```

If you want an ASCII program file, use SAVE or RE-SAVE. You'll normally use ASCII files for your programs only if you're using the same program on different types of computers.

Adding a Label

When you initialize a disk, HP BASIC gives it a *volume label* to help identify it. You can change this label with the PRINT LABEL statement.

For example, to label the disk in drive A of a personal computer as "FILES", type:

```
PRINT LABEL "FILES" TO ":CS80,1500,0" ↵
```

(If you don't specify the mass storage unit, the label is printed on the disk in the current unit.)

The label, up to six characters long, lets you identify the disk. For instance, this short section of code checks to see if you have the correct disk in drive A:

```
520 READ LABEL Label$ FROM ":CS80,1500,0"  
530 IF Label$ <> "FILES" THEN PRINT "Wrong disk!"
```

Programs vs. Data

Programs you store or save are sets of BASIC instructions that make the computer do some task. *Data* you put into mass storage can be numbers or ASCII characters or both. But it's just that – data and nothing more. It has meaning only when it's brought into the computer by a program and added, formatted, printed, plotted, or otherwise used for something.

You know how to store programs on disk. And you know the difference between programs and data. Now read on, as you learn how to put Emma Haredale's data on disk for her.

Create a Data File

For program storage, one statement (STORE or SAVE) does it all. In data storage, you create the data file separately, then open a path to the file, and finally output the data.

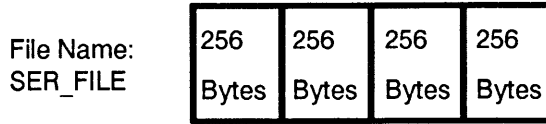
Here's the statement you'll use to create the data file:

```
90 CREATE BDAT "SER_FILE", 4
```

Before you continue, you need to learn a little more about files and file types.

All About Files

All information you place on mass storage is held there in *files*. A file, in turn, can have any number of *records*. A record is the smallest unit of data storage you can address.



4 Records

In the illustration the file name "SER_FILE" is made up of four records. Each record in this file has 256 bytes of memory. All records in a file have the same number of bytes, although different files can have different record sizes.

When you do a CAT of a mass storage unit, you can see information about all files on the unit. Here's an example:

```

:CS88, 1500
VOLUME LABEL: FILES
FILE NAME PRO TYPE REC/FILE BYTE/REC ADDRESS DATE TIME
SER_FILE BDAT 4 256 26 4-Apr-88 13:12
RAND_FILE BDAT 100 8 31 4-Apr-88 13:23
RAND_ROOTS PROG 4 256 35 4-Apr-88 14:21
MORTGAGE PROG 3 256 39 5-Apr-88 0:35
PANAMA PROG 2 256 42 5-Apr-88 0:36
ROLLDICE PROG 2 256 44 5-Apr-88 0:36
QUADRATIC PROG 3 256 46 5-Apr-88 0:37
MORNING PROG 5 256 49 5-Apr-88 0:38
GUESS_GAME PROG 2 256 54 5-Apr-88 0:38
SER_ROOTS PROG 3 256 56 5-Apr-88 0:44
MAGIC PROG 4 256 60 5-Apr-88 0:41
COUNT_DISP PROG 2 256 64 5-Apr-88 7:29
COUNT_PRNT ASCII 1 256 66 5-Apr-88 7:30
-

```

EDIT	Continue	RUN	SCRATCH	LOAD	LOAD BIN	LIST BIN	RE-STORE
------	----------	-----	---------	------	----------	----------	----------

File Name: Each file name is unique. Remember that file names are case-sensitive, though, so the file "RANDOM" and the file "Random" are not the same.

Pro Type: The type of file is next to its name. PROG and ASCII files hold programs. BDAT, ASCII and HP-UX files hold data.

Rec/File: Number of records in this file.

Byte/Rec: Number of bytes per record.

Address: The address is the beginning location of the file on the mass storage unit.

What Kind of File to Use?

When creating a file to hold data, you have to choose what type of data storage you want: BDAT, ASCII, or HP-UX.

- BDAT (binary data) files offer fast, efficient transfer of data, and serial or random access—you can get to the data in one big chunk or piece by piece. They let you choose the format—whether you want the data represented in an internal bit format or as ASCII bytes—and they usually take less memory than ASCII files. However, BDAT files aren't usable by other types of computers.
- ASCII files save all data as strings of ASCII characters, so they chew up a lot of memory. You can access the data serially only—in one big chunk—and outputting to the file or reading from usually takes more time than with BDAT files. But they do have one big advantage: data is in a Logical Interchange Format (LIF) used by many HP computers, so you can use the data file on different systems.
- HP-UX files are similar to BDAT files.

In general, BDAT files are the best choice for your data, unless you need to move it back and forth to other systems.

How Large a File?

Part of creating a file is determining how large it must be to hold your data.

Data in BDAT files: BDAT integer values require two bytes per number, while real values need the full-precision eight bytes per number.

ASCII strings use one byte per character, plus an additional four bytes. If the total number of characters is an odd number, one more "pad" byte is added to make it an even number.

The first four bytes in a string are used for a header that tells how long the string is.

Integer Numbers



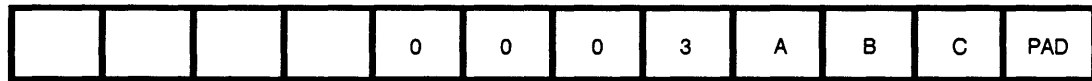
2 Bytes/Element

Real Numbers



8 Bytes/Element

ASCII Strings

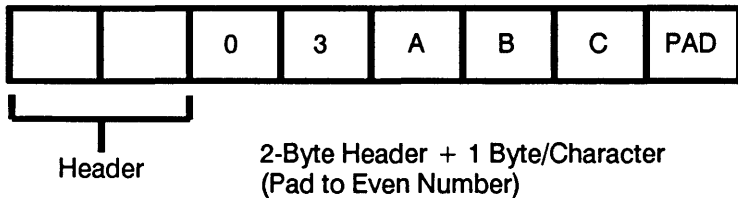


Header

4-Byte Header
+ 1 Byte/Character
+ Pad to Even Number

Data in ASCII files: In ASCII data files, every item – string, real number, or integer number – is stored as ASCII characters. The item uses one byte per character, with an additional two-byte header. If the number of characters is an odd number, one "pad" byte is added to make it even.

A real number stored in ASCII requires one byte for each character, plus one byte each for the sign, decimal point, exponent, exponent sign and exponent value.



Data in HP-UX files: HP-UX files are very similar to BDAT files. The difference is that string data in HP-UX files doesn't have a header, but does have a termination character that occupies one byte.

The chart below compares storage using different file types for some typical values of data.

Data:	In BDAT Requires:	In ASCII Requires:	In HP-UX Requires:
125 (Integer number)	2 bytes	6 bytes (2-byte header, 3 bytes for characters, 1-byte pad)	2 bytes
5.04 (Real number)	8 bytes	8 bytes	8 bytes
1.2345E + 10 (Real number)	8 bytes	14 bytes (5 bytes for characters, 6 bytes for sign, decimal point, etc., 2-byte header, 1-byte pad)	8 bytes
HELLO (String)	10 bytes	8 bytes	6 bytes
125 (As string data)	8 bytes	6 bytes	4 bytes

The CREATE Statement

With all this information about files and records and bytes under your belt, let's look at that CREATE statement once again:

```
90 CREATE BDAT "SER_FILE", 4
```

This statement creates a BDAT file named SER_FILE. The file has four records of 256 bytes each.

In the CREATE statement, you must name the file and specify the number of records. You can, if you like, specify a record length; if you don't, each record is automatically set at 256 bytes.

(HP-UX files are a special case. You can't specify a record length—records in these files are always 1 byte long. And if you try to specify a number lower than 256 for the number of records, BASIC automatically gives you 256 records.)

Look at these examples of CREATE statements:

```
100 CREATE BDAT "Data1", 5
```

Line 100 creates a BDAT file called Data1. The file contains 5 records of 256 bytes each.

```
CREATE ASCII, "WORDS", 2000, 32
```

This statement creates an ASCII file named WORDS. The file is 2000 records of 32 bytes each (64,000 bytes total).

If you don't specify ASCII or BDAT, CREATE produces an HP-UX file:

```
160 CREATE "File", 512
```

Line 160 creates an HP-UX file consisting of 512 records, each 1 byte long.

Consider the record length carefully if you have large files. Try to use integral powers of 2 (that is, 4, 8, 16, 32, 64, 128) up to a record length of 256 bytes. Above 256, use multiples of 256 (for example, 512, 1024, etc.).

Of Files HP-UX and DOS

As you might suspect from the name, HP-UX files are designed for sharing data between BASIC and the UNIX environment. But the truth is, you don't need UNIX to benefit from using this file type.

For instance, if you're using the BASIC Language Processor card in a Vectra or other personal computer, HP-UX files give you an added "plus" over other file types: transportability to DOS.

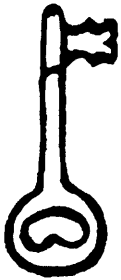
Here's how it can work:

1. You create an HP-UX file in HP BASIC.
2. You can use HP BASIC's OUTPUT statement to place string data in the file.
3. From a DOS word processor or spreadsheet program, you can read that data.
4. And, naturally, you can also read and write to the same file using HP BASIC.

For this application, HP-UX files are superior to ASCII files created using BASIC, because HP BASIC's ASCII files are burdened with additional information (headers and pad bytes).

Incidentally, all HP-UX files are write-protected for use in the DOS environment; this prevents you from accidentally corrupting an HP-UX file by writing to it with your word processor. To *write* to an HP-UX file from a DOS application, you'll have to change the read-only file attribute—perhaps with the Norton Utilities or another similar utility program.

Open a Path to the File



Whew! All that just to create the file. And before you can stick any data in it, there's more – you have to *open* the file.

You use the ASSIGN statement like a key, to open a path to the file for input/output (I/O). Here's the one we'll use for Haredale's file:

```
100 ASSIGN @Access_path TO "SER_FILE"
```

This statement opens an I/O path to the file SER_FILE. Now, when you actually output data to the file, you don't refer to the file at all – you just refer to the *path*, like this:

```
110 OUTPUT @Access_path; Root_array(*)
```

An I/O path name is just like a variable name (first letter a capital, all others lowercase, etc.). You can tell an I/O path name because it always has the "at" sign (@) in front.

Assigning an I/O path name to a file lets the computer "remember" the type of file and where it is. The I/O path is then used instead of the actual file name for input/output operations such as OUTPUT, ENTER, and TRANSFER.

If you don't mention the mass storage unit in an ASSIGN statement, the system looks on the current unit for the file. You can add the mass storage unit's specifier to ASSIGN a path name to a file in a different storage unit. For example:

```
700 ASSIGN @Datapath TO "DATA2:,1500,2"
```

Line 700 opens an I/O path to the file DATA2, located on drive C of an HP Vectra PC or other personal computer.

Output Data Along the Path

You've created a file and opened a path to it. The next step is to output data along the path. For this, you use the OUTPUT statement:

```
110 OUTPUT @Access_path; Root_array(*)
```

This statement instructs the computer to output all data from an array called "Root_array" along the I/O path called "@Access_path."

This is a *serial* output – all the data flows out in a big chunk – so you'll be able to bring it back in only with serial access.

When you OUTPUT data to a file, you should decide whether you will want serial access or random access to that data later.

Serial Access

Serial access means that the data transfer between your computer's memory and the mass storage unit goes very quickly. But to get any specific piece of data, you must bring all the data back into memory.

Random Access

In random access, you can get to any part of the data in the file. If you need to bring in all the data at once, though, random access makes transfer slower – a lot slower – than does serial access.

When you make the decision for serial or random access, you'll have to consider how the file is structured, and how the data is placed into the file.

Close the Path to the File

After you've opened a path to the file and output data to it (or even after you've entered data from a file), you should *close* the path. You close the path by assigning the I/O path name to an asterisk, like this:

```
180 ASSIGN @Access_path TO *
```

That's all there is to it! Line 180 closes the access path (called "@Access_path") to the file "SER_FILE."

There are other ways of closing an I/O path. A path is closed:

- When a program stops running or is affected by STOP, END, SCRATCH, EDIT, etc.
- When execution moves to a subprogram, or back to the main program from a subprogram.
- When the path name is reassigned – for example, to a different file.

It's good programming practice to always use a statement like the one in line 180 to close a path you've opened.

Cardinal Rule

Always use ASSIGN @___ TO * to close a path when you've finished moving data.

Entering Data from Disk

Now that you've figured out how to save Haredale's data on disk, how does she bring it back in? The answer is the ENTER statement.

As with OUTPUT, you don't ENTER data directly from a file. Instead, you must:

1. Open an I/O path to the file.
2. Use ENTER to bring data in.
3. Close the I/O path.

The code might look like this:

```
130 ASSIGN @New_path; TO "SER_FILE"  
140 ENTER @New_path; Read_array(*)  
150 ASSIGN @New_path; TO *
```

Line 140 reads data from the path called "@New_path" into an array called "Read_array."

If you output string data to the file, be sure to bring it back into string variables. If you used random output, bring it back the same way. Remember this very important cardinal rule:

Cardinal Rule 

ENTER it the way you OUTPUT it!

Now it's time to put Emma Haredale's square roots into a file for her.

Insert the disk with examples into the drive you've been using. Type:

```
LOAD "SER_ROOTS" _
```

Then remove the disk of example programs and replace it with your formatted data disk (which you named FILES).

To see a listing of the SER_ROOTS program, type:

```
LIST _
```

```
10 ! SER_ROOTS
20 OPTION BASE 1
30 PRINTER IS 1
40 MASS STORAGE IS ":",1500,0"
50 DIM Root_array(100),Read_array(100)
60 FOR I=1 TO 100
70 Root_array(I)=SQRT(I)
80 NEXT I
90 CREATE BDAT "SER_FILE",4
100 ASSIGN @Access_path TO "SER_FILE"
110 OUTPUT @Access_path;Root_array(*)
120 ASSIGN @Access_path TO *
130 ASSIGN @New_path TO "SER_FILE"
140 ENTER @New_path;Read_array(*)
150 ASSIGN @New_path TO *
160 FOR I=1 TO 100
170 PRINT "The square root of";I:"is";Read_array(I)
180 WAIT .1
190 NEXT I
200 END
```

Now run the program.

Note



This program creates a file on a disk in drive A (that is, on ":", 1500, 0"). If you want to use a different drive, you'll have to change the MASS STORAGE IS statement in line 40 so it refers to your drive.

1. Make sure there's an initialized disk in drive A (or the drive you're using).
2. Type:

```
RUN_↵
```

The program takes a few moments to create the file and put data on the disk. Then it brings the data back and prints the square roots of numbers from 1 to 100:

```
The square root of 1 is 1
.
.
The square root of 99 is 9.94987437107
The square root of 100 is 10
```

If the file `SER_FILE` is already on the disk, you don't need to `CREATE` it; in fact, you'll get an error like this:

```
Duplicate file name
```

In this case, just "comment out" line 90 by inserting an exclamation point that turns that line into an unexecuted remark:

```
EDIT 90_↵
```

```
90 ! CREATE BDAT "SER_FILE", 4
```

How it works: Lines 20-50 set up mass storage and the arrays you need:

```
20 OPTION BASE 1
30 PRINTER IS 1
40 MASS STORAGE IS ":CS80,1500,0"
50 DIM Root_array(100), Read_array(100)
```

Lines 60-80 are a loop that calculates the square roots of the numbers from 1 to 100 and puts them in an array called "Root_array":

```
60 FOR I=1 TO 100
70 Root_array(I)=SQRT(I)
80 NEXT I
```

Line 90 creates a file of four 256-byte records – more than enough to hold 100 real numbers. Then lines 100-120 open a path to the file, output the numbers in the array, and close the path:

```
90 CREATE BDAT "SER_FILE", 4
100 ASSIGN @Access_path TO "SER_FILE"
110 OUTPUT @Access_path; Root_array(*)
120 ASSIGN @Access_path TO *
```

That's the end of storing data into the disk. The next section of code brings the data back. It uses a different path and puts the data into a different array. In fact, the code below could be part of another program, or on a different computer entirely – just as long as the file "SER_FILE" was available on the current mass storage unit.

```
130 ASSIGN @New_path TO "SER_FILE"
140 ENTER @New_path; Read_array(*)
150 ASSIGN @New_path TO *
160 FOR I=1 TO 100
170 PRINT "The square root of";I:"is;Read_array(I)
180 WAIT .1
190 NEXT I
```

A Random Example

What about random access? As luck would have it, your disk of examples also contains an example of a random-access file. It's similar to the earlier example, SER_ROOTS, but since it's for random access, it's called RAND_ROOTS. First, put the examples disk in your disk drive, and load and list the program.

```
LOAD "RAND_ROOTS" ↵
LIST ↵
```

```

10  ! RE-STORE "RAND_ROOTS"
20  OPTION BASE 1
30  PRINTER IS 1
40  MASS STORAGE IS ":CS80,1500,0"
50  DIM Root_array(100), Read_array(100)
60  FOR I=1 TO 100
70  Root_array(I)=SQRT(I)
80  NEXT I
90  CREATE BDAT "RAND_FILE",100,8
100 ASSIGN @Access_path TO "RAND_FILE"
110 FOR I=1 TO 100
120 OUTPUT @Access_path,I;Root_array(I)
130 NEXT I
140 ASSIGN @Access_path TO *
150 ASSIGN @New_path TO "RAND_FILE"
160 FOR I=1 TO 100
170 ENTER @New_path, I;Read_array(I)
180 NEXT I
190 FOR I=1 TO 100
200 PRINT "The square root of";I;"is";Read_array(I)
210 WAIT .1
220 NEXT I
230 END

```

Now substitute an initialized disk for the write-protected examples disk, and run the program. (You may need to change line 40 to "point to" your mass storage unit.)

You'll probably have enough time to make coffee, as each random root is laboriously output to its own record on the disk:

RUN_↓

How it works: The CREATE statement in line 90 creates a binary data file of 100 records with 8 bytes per record—just the right size for 100 real numbers.

The key to random output is the OUTPUT statement in line 120:

```
120 OUTPUT @Access_path, I;Root_array(I)
```

The "I" after the comma is the *record number*. It tells where this piece of data will be stored on the disk.

The ENTER statement in line 170 also uses the record number to get to the data on the disk. (Read it like you wrote it, remember?)

Although it's faster to bring the data into an array, then use it, you can use the record number to get to any piece of random data in mass storage. Try this mini-program:

```
SCRATCH_┘  
EDIT_┘
```

```
10 ASSIGN @Mypath TO "RAND_FILE"  
20 ENTER @Mypath, 20; What_is_it  
30 PRINT What_is_it  
40 ASSIGN @Mypath TO *  
50 END
```

RUN_┘

When you run this, line 20 reads the data in record 20 to the variable `What_is_it`, which is then printed:

4.472135955

Random access of data, then, is slow – but surgically precise.

Review Quiz

1. You have a lot of data in the form of integers to store on disk. What's the best type of file for maximum speed and efficiency?

What files allow the use of data on other computers?

Which files provide random access of data?

2. How much mass storage is needed for these quantities stored in a BDAT data file? In an ASCII file?

Quantity	Bytes in BDAT	Bytes in ASCII
6.02E + 2 (Real)	?	?
3.14159 (Integer)	?	?
NOW IS THE TIME (String)	?	?

3. Put these statements in the order you'd execute them in a program to put data on a disk:

```
ASSIGN  
MSI  
OUTPUT  
CREATE
```

4. Write a `CREATE` statement for an ASCII data file called `CITIZENS`. The data will be 1000 names of citizens. No name will be longer than 30 characters.

Laboratory Exercise

In this exercise, you fill arrays with random data and ASCII data, and benchmark them against a serial array.

Load the shell program `LAB10_SHL` from your disk of examples. This shell contains code to generate 100 integer numbers, as well as program lines that use the `TIMEDATE` function to time how long it takes for storage and retrieval. (You'll learn more about `TIMEDATE` in lesson 29; for now, you can just watch as this function calculates the time needed for different operations.)

Use comments within the shell to help you write the code.

There's also a solution on the examples disk. It's the program `SOL_LAB10`. But try to get the program to work yourself, before looking at the solution.

Part 2

Instrument Control with HP-IB

This part consists of lessons 11-20. Here you'll learn about HP-IB and the fundamentals of using this interface to control electronic instruments and test equipment.

Before beginning this part, you should already be familiar with programming in BASIC. Perhaps you've just completed Part 1, Basic BASIC Programming, and are itching to continue. Or maybe you already have a good BASIC background and are beginning the course here.

Either way, turn the page now. And start controlling those instruments!

Introduction to HP-IB

If you worked through Part 1 (lessons 1-10) of this course, congratulations! You're ready to begin learning how to put all that programming knowledge to work controlling devices on HP-IB.

If you already have a BASIC programming background and skipped Part 1, welcome! You should have no trouble with Part 2 (lessons 11-20), especially if you use the index to refer back to Part 1 for anything you don't understand.

This lesson is an introduction to HP-IB. In it you'll learn about:

- What an interface is.
- The interfaces in your computer.
- Bits and bytes in BASIC.
- Roles in HP-IB: talker, listener, controller, system controller.
- HP-IB signal lines: data, handshake, bus management.

You usually don't need to know how HP-IB works, especially if you're using it with Hewlett-Packard instruments and HP computers. Still, to guarantee you make the best use of HP-IB with HP BASIC, be sure to work through this lesson.

HP-IB: A Short History

Adding interface capabilities to electronic instruments and measuring devices – so they could "talk" to computers and other instruments – seemed like a wonderful thing in the early 1970s. But engineers and designers soon found themselves with a major-league headache: nothing worked with anything else.

Oh, sure, the ABC company's multimeter worked with a microcomputer, all right. But it didn't use the same interface as the XYZ company's signal generator, nor the LMN company's power supplies. Every solution was unique.

And even though countless hours were being chewed up developing ways for devices to communicate, there was another problem, too. As soon as one of these rudimentary interfaces was developed, everybody – manufacturers, customers, and designers – tried to leapfrog to applications far beyond its capabilities.

Enter the International Electrotechnical Commission (IEC). In 1972, this august body began to look for a single interface that could be applied worldwide, a standard that would be usable with a wide variety of products.

Among the plethora of standards proffered by manufacturers around the world was one developed at the Hewlett-Packard Company of Palo Alto, California. Originally designed to link the company's test equipment, this standard was known as the Hewlett-Packard Interface Bus (HP-IB). It was chosen as a model by both the IEC and by a United States subcommittee from the Institute of Electrical and Electronics Engineers (IEEE).

In the next two years, several refinements were added to the interface: extended addressing beyond the primary addresses, a capability to poll many instruments at once, and a mechanism allowing control to be passed from one device to another. Finally, in 1974, the IEEE Standards Board approved what is now known as IEEE Std 488.

The standard was revised in 1978, and a supplement added in 1980. Meanwhile, the international standard, known as IEC 625-1, was published in 1979. No matter what you call it – HP-IB, IEEE-488, GPIB, or IEC-625 – all forms of this bus are essentially the same. HP-IB is truly an international standard.

What Is an Interface?

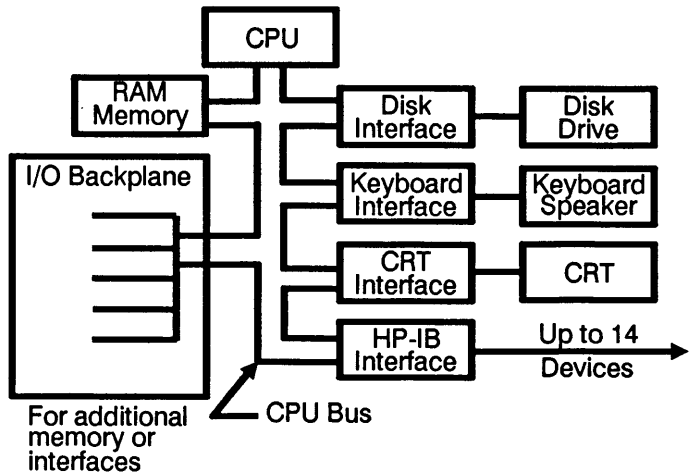
Simply put, an *interface* is an intermediary between incompatible systems.

When you drive an automobile, the steering wheel, brake and accelerator pedals, and gear shift lever are interfaces between you and various systems on the car. Each interface translates your foot or hand motion into something that can be understood and used by part of the automobile.

A computer interface, such as HP-IB, makes it possible for different systems, or different parts of a computer, to talk to each other – and to be understood.

Interfaces in Your Computer

Even if you haven't used HP-IB, you've already used interfaces. You see, within your computer, each of the different parts requires an interface to connect it to other parts, and to the CPU (the central processing unit – the computer's "brain").



A disk interface connects the disk drive, a keyboard interface links the keyboard, there's yet another interface for the CRT display, and so on.

Chances are, you plug other interface boards into your computer. You might use an RS-232-C interface to talk to a modem, or a Centronics parallel interface to communicate with a printer. You can even add additional HP-IB interface cards. (As long as they don't have the same interface select code as your internal HP-IB. More about select codes later.)

Although it was originally developed at Hewlett-Packard, HP-IB is an industry-standard interface. As something called "GPIB" or "IEEE-488", it's used by—and available from—many different manufacturers. It's not the only interface you can use, either. Look at these examples of other standard interfaces found in the computer and electronics industries:

Interface Name	Primary Use
Centronics	Controlling printers
RS-232-C	Controlling printers, external disk drives, data communications
BCD	Controlling instruments
Datacomm	Data communications
GPIO	Controlling instruments, specialized applications

You may find HP-IB and one or all of these other interfaces in the same computer.

Compatibility: Four Vital Areas

As an interface, HP-IB has to make sure all devices hooked to it are compatible in four areas:

1. **Mechanical:** It must physically connect all devices. All connectors must fit.
2. **Electrical:** It has to meet voltage and current requirements of all devices.
3. **Functional:** Since data formats may not be the same, intelligence within HP-IB converts data from the computer's internal representation to that of other devices.
4. **Timing and Handshake:** It must move data between devices at a known rate, or with a "handshake" after each data item that acknowledges receipt.

The HP-IB

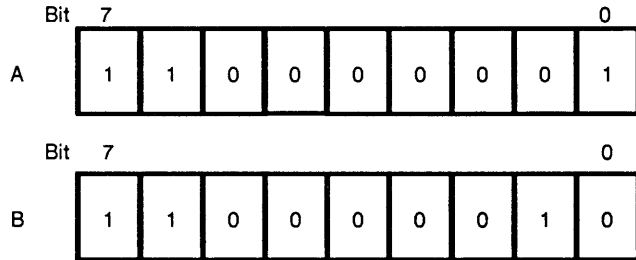
The Hewlett-Packard Interface Bus, or HP-IB, is Hewlett-Packard's version of a 1978 standard known as IEEE-488. (IEEE means *Institute of Electrical and Electronics Engineers*.)

Computer Bits and Bytes

Your computer is a *digital* computer. That means it sees everything, including words, numbers, commands, and statements, as collections of ones and zeros. These ones and zeros are called "bits."

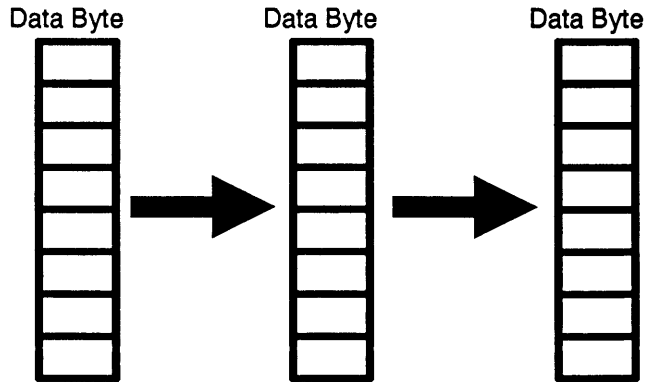
For easy movement and manipulation, data bits in BASIC are formed into collections known as *bytes*. In this scheme, one byte is eight bits.

A single character occupies a single byte. Look at the difference between the bits for an ASCII "A" and a "B":



The difference is only in bits 0 and 1. (In reality, bit 8 isn't used for the character at all. It's called the *parity bit*. But it's still part of the byte.)

In HP-IB, message transfer is *bit-parallel, byte-serial*. This means that one 8-bit data byte is sent at a time.



Bit-parallel, byte-serial transmission means data transfer is quite fast.

HP-IB Features

Here are a few features of HP-IB you'll want to know:

- Maximum data rate on HP-IB is 1 megabyte per second. (Typical rates are 5 to 20 kilobytes per second.)
- You can pass control from one controller to another on the same bus.
- You can have up to 15 devices (including controller) on the bus.
- It's for short-distance control. Maximum total cable length is 20 meters, although you can purchase extender circuits.

What's the Difference?

You will see references to GPIB, to IEEE-488-1978, to IEC-625, or to HP-IB. These are *all the same thing*. The four areas of compatibility—electrical, mechanical, functional and operational—are all the same. (IEC-625 is different mechanically, using different cables and connectors.)

Cardinal Rule

HP-IB is electrically the same as GPIB, IEEE-488-1978 and IEC-625.

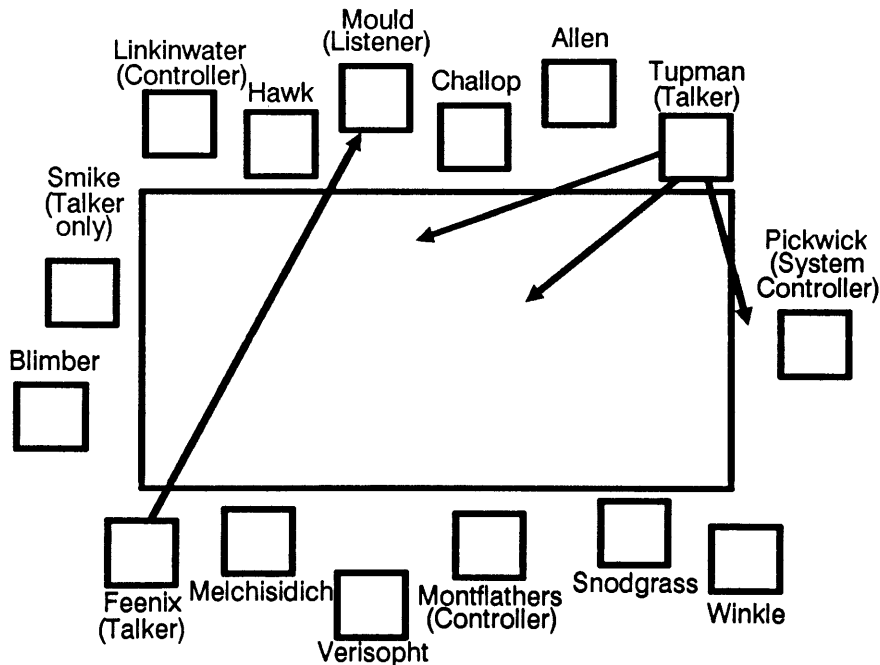
GPIB and HP-IB are the same, but the interface known as GPIO is different.

On the Bus

If your test setup has only one instrument and a computer, you needn't worry much about what kind of activity you'll find on HP-IB. As you get into more complex systems, though, you'll need to understand the different roles that devices play in HP-IB. To visualize, think of HP-IB as a club meeting.

Example: The Pickwick Club: As president, Mr. Pickwick brooks no nonsense at the Pickwick Club. He limits membership to 14 persons, plus himself. There are four ironclad rules:

1. Pickwick controls the meeting. He decides who is permitted to talk.
2. There can be only one talker. All others may be listeners.



3. The strangest rule is that Pickwick also controls who listens. Nobody can listen unless Pickwick says so.
4. Pickwick can allow other persons to control who talks and who listens. But as president, Pickwick always asserts control at the beginning of the meeting; and he sometimes reasserts control right in mid-meeting.

Today's meeting is typical. Pickwick tells Feenix to talk to Mould. Then Pickwick tells Mould to listen. Feenix spews out story after story to Mould, until he has no more information.

Now Pickwick lets his favorite, Montflathers, control the meeting. Montflathers directs Tupman to talk and everyone (including herself and Pickwick) to listen.

Within the club, there are limitations on what each member can do. Smike, for example, can only talk; he can't listen. Only Pickwick, Montflathers and Linkinwater are capable of controlling the meeting. And, of course, only Pickwick can be the overall controller.

Roles on HP-IB

HP-IB is surprisingly similar to the Pickwick Club. It also can have up to 15 "members" – devices on the bus. Instead of names, though, each device is identified with an address of 0 through 30.

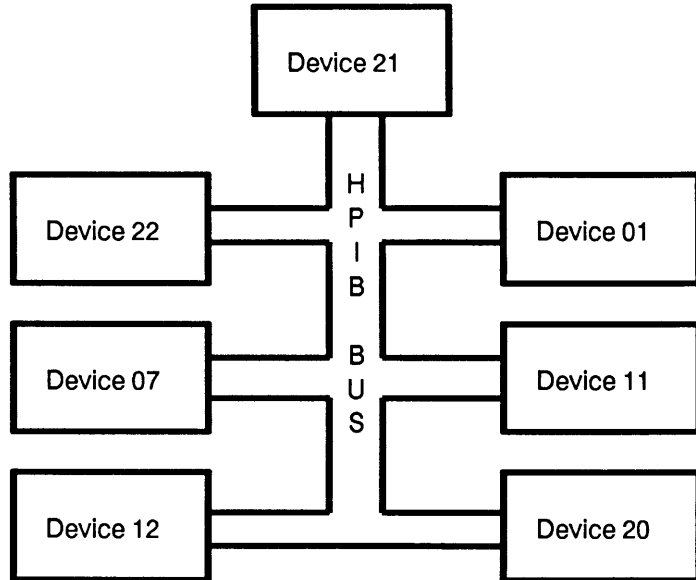
Each device on the bus can act as *at least one* (and often more) of these:

Listener: Can receive data over the bus from other devices.

Talker: Can transmit data (but not commands) over the interface to other devices. There can be only one talker active at once.

Controller: Can specify the talker and listeners (including itself) for an information transfer.

System Controller: The master controller in a multi-controller system. The system controller takes control of the bus when power is turned on or when something goes wrong with normal bus operations.



In systems with more than one controller, only one controller can be active at one time. The active controller can *pass control* to another controller; but only the system controller can *assume control*.

What a Device Can Do

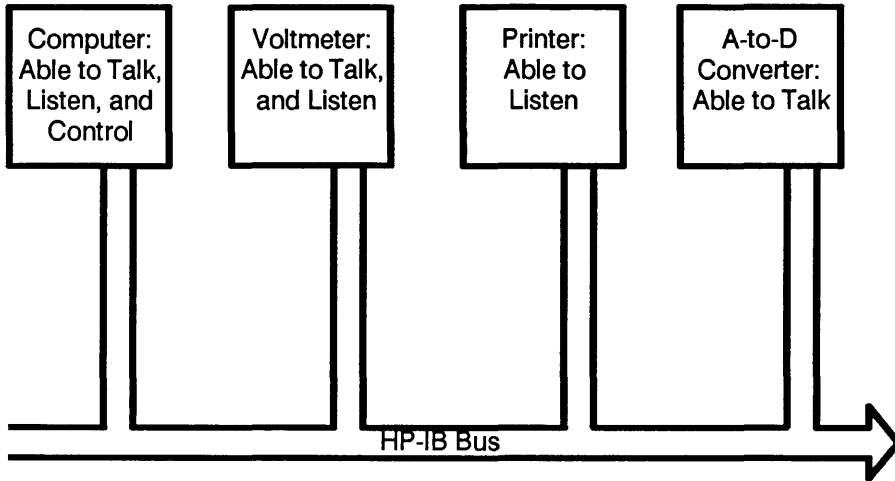
Depending on the instrument or other device, you'll find different capabilities.

A *computer*, of course, can be a talker, a listener, or a controller.

A *voltmeter* can be a talker or listener, but probably not a controller.

A *printer* is an example of a device that can only be a listener.

An *A-to-D converter* can be a talker, but not a listener or a controller.



All data transfer takes place at the rate of the slowest listener. That's why the ability to choose listeners is so important – because you can eliminate slow listeners (such as printers) when they're not needed, and make data transfers from a talker much more quickly.

Today's electronic instruments, of course, are very powerful. You'll find instruments that not only can act as a talker or listener, but that can even be a controller or system controller.

The system controller doesn't have to be a computer. A "smart" instrument such as the HP 8753A RF Network Analyzer, for instance, can act as a system controller on HP-IB.

The Bus Lines

HP-IB has 16 lines, used for data, for handshaking, and for bus management.

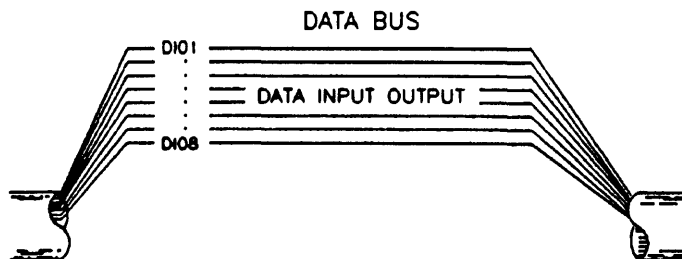
If you're working with a computer and one or two instruments, you probably won't care what's on these lines. Even with more instruments, you can write successful, working control programs without so much as a scintilla of knowledge about them.

As your programming becomes more sophisticated, though, and as you try to speed up your programs, you'll want to know every detail of how HP-IB program lines operate. So read on now—you'll thank yourself later.

Data Lines

As you know, HP-IB is a bit-parallel, byte-serial interface. In order to send an entire 8-bit byte of data at one time, eight separate data lines are used. These are numbered DIO1 through DIO8.

All *data* communication takes place between the talker and the listener (or listeners). The end of data is usually indicated by a CR/LF (carriage return/line feed). When the listener receives a CR/LF, it assumes all data has been sent.



Note

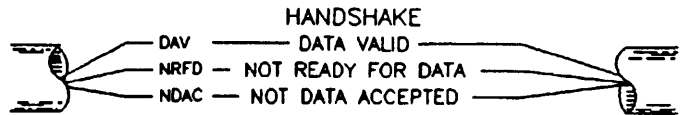


Instruments are picky creatures. Usually, when an instrument sees a CR/LF in a data block, it assumes that's the end of the data—whether you intended it or not. Pay attention to CR/LF in your data, and you'll be a giant step ahead when debugging.

Handshake Lines

The so-called "handshake" coordinates the passing of data from talker to listener.

HP-IB uses a three-wire handshake to guarantee the integrity of data, even among devices operating at different transfer rates. Every transmitted byte undergoes a handshake, so there's no way you can lose data. The transfer takes place at the rate of the slowest device.



Each line is always in one of two states: true or false.

The handshake lines are:

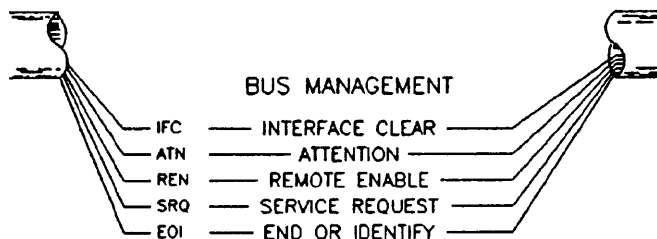
DAV: Data Valid. Controlled by the talker or controller. When true, it indicates data is available.

NRFD: Not Ready For Data. Controlled by the listener, or devices receiving commands. When false, it indicates the device can receive data.

NDAC: Not Data Accepted. Controlled by listener or devices receiving commands. When false, it indicates data has been accepted.

Bus Management

There are five lines for management of the bus. They are IFC, ATN, REN, SRQ and EOI.



Here's how the bus management lines operate:

IFC: Interface Clear. Controlled *only* by the system controller. It stops all activity on the bus.

ATN: Attention. This line is controlled by the active controller, and specifies what's being sent on the data lines. If ATN is true, it's a command. If ATN is false, data on the lines is just that – data.

The Attention line separates data from commands.

REN: Remote Enable. Controlled by system controller. When REN is true, devices respond to remote program (computer) control. When it's false, instruments and other devices are in local operation – that is, controlled from the instrument's front panel, not from a computer.

SRQ: Service Request. Controlled by any device *except* the controller. When this line is true, it signals to the controller that a device needs attention. (For example, if there's an error condition, data to send, etc.)

EOI: End or Identify. This line is controlled by the talker when ATN is false (data mode); or by the controller when ATN is true (command mode). EOI indicates the last byte of data in a multibyte sequence (that is, the end of data) if controlled by the talker. If the system controller has turned EOI on and ATN is true, it indicates the controller is doing a parallel poll. (A parallel poll is a way of checking the status of devices on the bus.)

With few exceptions, you won't need to worry about any of these lines until you get to the more sophisticated techniques for instrument control in part 3 of this course. In this part of the course, the lines are almost transparent. But you'll be reminded of them from time to time, and you'll need to know about them when you yourself want to check instrument status.

Review Quiz

1. True or False: HP-IB is a byte-parallel, bit-serial interface?
2. What's the difference between HP-IB and IEEE-488?
3. In HP-BASIC, how many bits are there in one byte of data?
4. Imagine for a moment that you are a listener on HP-IB. Suddenly your lines DIO1-DIO8 begin to fill with data! You quickly check ATN and find it's on (true). Do you interpret the incoming signal as data or as a command?
5. Name two ways to detect the end of a data block.

Installing HP-IB Hardware

This lesson explains all about HP-IB hardware. It explains how to physically connect devices together, and what signals are available on an HP-IB cable. In this lesson, you'll learn about:

- How to identify HP-IB devices.
- How to determine HP-IB capability.
- Setting device addresses.
- What cables to use for HP-IB.
- How to connect devices together.
- Star and linear arrangements.
- Cable lengths.

Identifying HP-IB Devices

In order to be connected via HP-IB (or GPIB), an instrument or computer must have an HP-IB connector.

It's a 24-pin female connector. Not just any 24-pin connector will do, either. The computer or instrument must have this special HP-IB connector.

When you locate the connector, make sure the screws are colored black. (They may be colored black or silver.) There are very good reasons why.

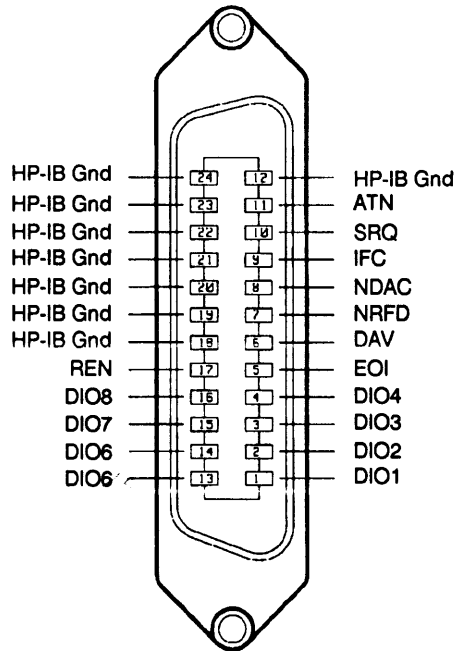
- Black connectors are for HP-IB; they have metric threads.
- Silver connectors have English threads. They're not used for HP-IB (except for some very early models of HP-IB instruments). Don't try to use a black-connector cable on a silver-connector device, or vice versa. Although the *signal lines* and pins are the same, the threads are different, and you might destroy both connectors.

Cardinal Rule 

Don't try to mate silver with black connectors. Nearly all HP-IB connectors are black.

Pins on the Connector

The pins and their corresponding lines on an HP-IB connector are always the same. Every HP-IB (or GPIB) instrument, computer, printer, or other device has exactly the same set of pins. They look like the illustration.



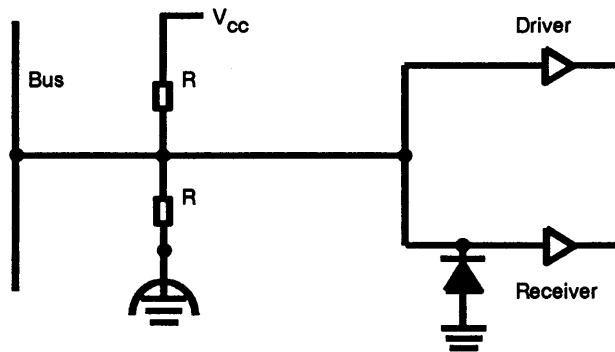
Inside the "HP-IB-Capable" Device

Within instruments, and other devices, there is a wide range of HP-IB or GPIB capabilities. Some devices make use of every line of the HP-IB interface. Others use just a few.

In order for a device to be "HP-IB-capable", all it *must* have are:

- An HP-IB connector
- A termination for each line of the interface.

A typical termination has a driver and a receiver, along with a pair of resistors connected to V_{CC} and ground.



Until you examine a device's manuals, or try it out, you don't know what capabilities it has.

Determining What an Instrument Can Do

Although all HP-IB devices have the same signal lines, they cannot all perform the same functions on the interface. Interface functions are capabilities that *may* be in an HP-IB device or instrument.

Depending on the device, this is how you'll find the capabilities shown (on the device's cabinet, or in its manual):

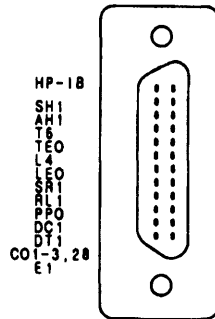
Designation	Meaning	Capability
T, TE	Talker or Extended Talker	Required for a device to be a talker. (An extended talker can talk to other devices at its address.)
L, LE	Listener or Extended Listener	Required for a device to be a listener.
SH	Source Handshake	Required to properly transfer a multiple message.
AH	Acceptor Handshake	Required to guarantee proper reception of a multiline message.
RL	Remote/Local	Can select between two sources of input: <i>local</i> corresponds to front-panel controls, and <i>remote</i> corresponds to input from the bus.
SR	Service Request	Can asynchronously request service from the controller.

Designation	Meaning	Capability
PP	Parallel Poll	Device can uniquely identify itself if it requires service and the controller is requesting a response. Differs from SR (service request) in that it requires the controller to periodically conduct a parallel poll.
DC	Device Clear	Device can be initialized to a pre-defined state. The effect of this command is described in the device's operating manual.
DT	Device Trigger	Device can have its basic operation initiated by the talker on the bus.
C	Controller	Device can send addresses, universal commands to other devices on the HP-IB. It may also be able to conduct polling to determine devices requiring service

Designation	Meaning	Capability
E	Drivers	This code describes the type of electrical drivers used in a device. E1 is open collector, E2 is tri-state.

Remember, these are just *possibilities*. You won't find all these capabilities in every instrument.

To find out what functions a device can perform, look next to the HP-IB connector or in the manual.



The example shown is for an HP 8340B Synthesized Sweeper.

On the Sweeper's cabinet, next to the HP-IB connector, is a list of codes that show the instrument's interface capabilities; each capability is shown with a number after it.

The number 0 means "not capable." The number 1 means "capable." Numbers above 1 show how much capability.

The letters and numbers are called "mnemonics" (pronounced "ni-mon-iks"). They're a kind of shorthand notation to show you the device's capabilities. The mnemonics you see are part of a set of standard mnemonics for IEEE Standard 488-1978.

You don't need to worry about the specific details of each of these mnemonics right now. There's a complete list of meanings in appendix B, and you'll learn more about them as you work through this course. However, so you'll understand how mnemonics show an instrument's capabilities, here's what the mnemonics on the sweeper mean:

Mnemonic	Meaning
SH1	Source Handshake: Complete capability.
AH1	Acceptor Handshake: Complete capability.
T6	Talker: Capable of basic talker, serial poll, and unaddress if my listen address (MLA).
TE0	Talker, Extended address: No capability.
L4	Listener: Capable of basic listener and unaddress if my talk address (MTA).
LE0	Listener, Extended address: No capability.
SR1	Service Request: Complete capability.
RL1	Remote Local: Complete capability.
PP0	Parallel Poll: No capability.

Mnemonic	Meaning
DC1	Device Clear: Complete capability.
C0, 1-3, 28	Controller capability options: C0, no capability. C1, system controller. C2, send interface clear (IFC) and take charge. C3, send remote enable (REN). C28, send interface messages.
E1	Electrical: Electrical specification indicating open collector outputs.

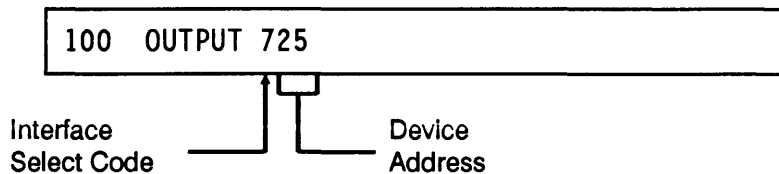
Don't worry too much about such arcane words as "my talk address" and "remote enable" now – you don't really need to know this level of detail until you get into part 3 of the course.

Setting Addresses

Once you've determined what HP-IB capabilities an instrument or other device has, the next step in installation is to set the HP-IB *address* of the instrument.

First, make sure you know the difference between the instrument *address* and the interface *select code*.

Here's how they appear in a program statement:



Interface Select Code

The interface select code for HP-IB is factory set at 7; and it usually remains set at 7.

Remember, the computer sees all of its components, including CRT display, keyboard, disk drives, etc., as interfaces. Each interface in the computer, whether internal or external, has a unique select code. You're already familiar with some of them.

Look at these examples of typical interface select codes

Select Code	Interface
1	CRT display(alpha)
2	Keyboard
3	CRT display (graphics)
9	Serial interface
7	HP-IB interface

Each interface must have a unique select code. Some codes, such as those of the CRT and keyboard, are fixed. You can change other select codes, however, usually by means of a switch or jumper on the interface card.

You sometimes specify the interface select code, with a statement such as:

```
PRINTER IS 1
```

or

```
LIST #26
```

The select code tells the computer where to look for the interface. The select code for each interface is *fixed*; it can't change unless you do it with hardware.

Unless you're really getting fancy, with more than one HP-IB interface connected, it's best to leave HP-IB at its factory setting of 7.

Device Address

Although the select code is enough to identify a serial or parallel interface, you need to be more specific to identify devices on HP-IB. Each device – whether it's an instrument, a printer, a plotter, or something else – must have a unique *address*. Even if there's only a single device on the bus, you'll still need to know its address.

On HP-IB, you can assign addresses of 00 through 30. You usually do this with a small switch located inside the instrument or on its rear panel; or by manipulating front panel controls.

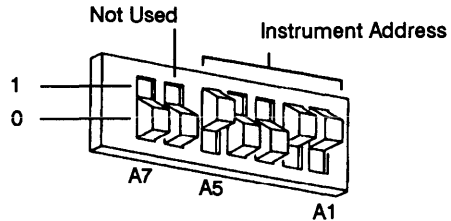
The device address is set at the factory, so you usually don't have to change it. The only time you need to change it is if more than one instrument or device *on the same bus* has the same address.

Reading a Device Address

The device address is factory-set, and you can find this information in the manual. If all you have is the switch itself to go by, remember this: the *address* is determined by *five* switches. Sometimes they're labeled A0-A4, sometimes A1-A5, and sometimes just ADDR.

If there are other switches, they determine other capabilities.

Here's an example:



The address switches are "binary-weighted." This means that the switch farthest right (A1 in the illustration) carries a "weight" of 1 if on, 0 if off.

A2 is 2 if on, 0 if off.

A3 is 4 if on, 0 if off.

A4 is 8 if on, 0 if off.

A5 is 16 if on, 0 if off.

To determine the decimal value, you just add up the numbers.

The switch shown above adds up like this:

Position	Value
A1	1
A2	+2
A3	+0
A4	+0
A5	+16
	Total = 19

The switch is set to address 19.

In this example, the TALK ONLY/ADDRESSABLE switch must be set to ADDRESSABLE if you want the instrument to be programmable via HP-IB.

Some instruments have more than one HP-IB address. The HP 8753A Network Analyzer, for instance, occupies *two* addresses: one for the instrument, and another for the CRT display.

Multiple-address devices often have fewer switches; so four switches (A2-A5) set the addresses. The setting of A1 doesn't matter, because the instrument uses a pair of sequential addresses (for example, addresses 12 and 13, or 18 and 19).

You'll find a complete list of HP-IB switch settings and addresses elsewhere in this lesson.

Secondary Addresses

Besides primary addresses (00-30), many devices can also have secondary addresses, up to six per device. This lets you address, say, a particular card or register in the device.

Changing a Device Address

If you've determined that, yes, you really need to change a device address, here's how to do it:

1. Turn off power to the instrument.
2. Set the switch to the new address.
3. Turn on power to the instrument.
4. Mark the front of the instrument with the new address so you won't forget it when you begin programming.

Not all instruments need a switch to set the HP-IB address. When you turn on the HP 8757A Scalar Network Analyzer, for instance, it shows you the current HP-IB address right on its CRT screen. To change the address (to 16, say) you press number keys, then press these keys on the front panel:

```
[LOCAL] 8757 16 [ENT]
```

HP-IB Addresses and Switch Settings

This chart shows settings for switches A1-A5, and the primary HP-IB addresses.

Address Switch Setting					Decimal HP-IB Address
5	4	3	2	1	
0	0	0	0	0	00
0	0	0	0	1	01
0	0	0	1	0	02
0	0	0	1	1	03
0	0	1	0	0	04
0	0	1	0	1	05
0	0	1	1	0	06
0	0	1	1	1	07
0	1	0	0	0	08
0	1	0	0	1	09
0	1	0	1	0	10
0	1	0	1	1	11
0	1	1	0	0	12
0	1	1	0	1	13
0	1	1	1	0	14
0	1	1	1	1	15
1	0	0	0	0	16
1	0	0	0	1	17
1	0	0	1	0	18

Address Switch Setting					Decimal HP-IB Address
1	0	0	1	1	19
1	0	1	0	0	20
1	0	1	0	1	21 (Always controller)
1	0	1	1	0	22
1	0	1	1	1	23
1	1	0	0	0	24
1	1	0	0	1	25
1	1	0	1	0	26
1	1	0	1	1	27
1	1	1	0	0	28
1	1	1	0	1	29
1	1	1	1	0	30
1	1	1	1	1	31 (Untalk or unlisten)

A few points:

The computer's factory-set address is 21, and you shouldn't try to use this for an instrument address.

Address 31 isn't really an address, but rather "untalk" or "unlisten." You'll learn more about these in part 3 (lesson 21) of this course.

What Cables to Use

Be sure to use *only* cables specifically designed for HP-IB, IEEE-488-1978, or IEC-625-1.

The standards IEEE-488 (that is, HP-IB) and ANSI MC1-1 use a 24-pin connector and cable. This is what you'll find in North America.

As an example, here are Hewlett-Packard part numbers for some typical HP-IB cables:

Part Number	Length
HP 10833A	1m (3.3 ft.)
HP 10833B	2m (6.6 ft.)
HP 10833C	4m (13.2 ft.)
HP 10833D	0.5m (1.6 ft.)

Make sure your cables and connectors are either all-black (metric fittings) or all-silver (English fittings) in color. Don't connect black screws to silver – you'll damage the hardware.

IEC 625-1 requires a 25-pin cable and connectors. Unfortunately, the 25-pin connector used for IEC 625-1 is the same used for the RS-232-C interface.

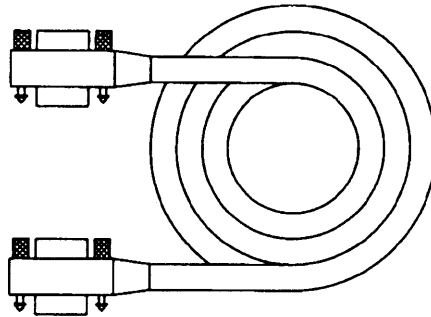
But, although it's easy to do, make sure you don't connect an RS-232-C circuit to an IEC 625-1 instrument. This connection can *damage the instrument!*

Caution



Don't mistakenly connect IEC 625-1 instruments to RS-232-C circuits.

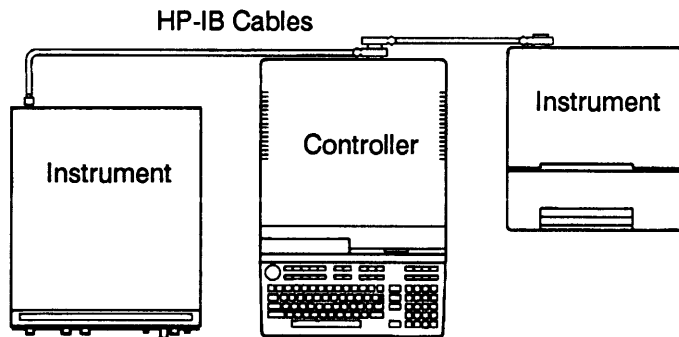
Standard HP-IB cables and connectors look like this:



Some "captive" cables on HP-IB devices differ slightly.

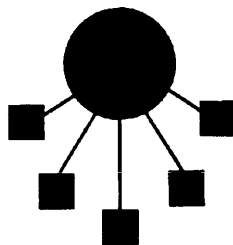
How to Connect Devices

You now have the instrument addresses set and the front panel of each instrument labeled with its address. (Don't you?) You have the cables you need. Now it's time to connect everything together.

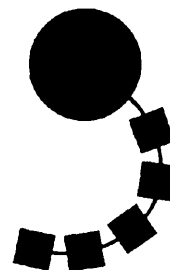


The cable connectors snap together easily. Connectors are designed with a male connector on one side and a female on the other. So you can "piggyback" more than one cable on a single instrument connector.

Star



Linear



You can connect instruments in a *star* pattern around the system controller. (The controller is usually a computer, but it can also be a "smart" instrument.) Another scheme is to connect everything in a *linear* pattern: that is, in a line.

The star configuration keeps cable lengths shorter. As you'll see when you start making connections, though, it becomes awkward to hook more than four cables to a single instrument or computer connector. Lumping devices together in a star can also produce high capacitance, which may create transmission errors.

Note



The linear configuration often means your cables must be longer, but it gives you more control of capacitance for error-free transmission.

You're not limited to one of these two schemes. You can connect instruments in combinations of star and linear patterns.

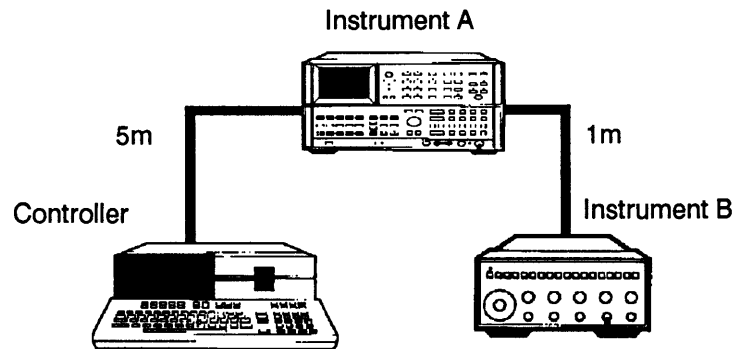
What About Cable Length?

You can connect as many devices together as you want with cables of any length, *except* that you're restricted to the *lesser* of:

- An average length of 2 meters per device.
- A total length of 20 meters.

The length of cable between any two devices doesn't matter, as long as these two rules are met.

Suppose you had a linear arrangement as shown here, with a controller and two instruments.



If the cable between the two instruments is 1 meter long, the cable from the controller can be as long as 5 meters. (3 devices x 2 meters each = 6 meters total.)

What happens if you exceed these lengths? You'll probably start to see more errors, and some commands may not give you the results you desire.

Keep Those Instruments On!

If you're tempted to turn off HP-IB instruments to save electricity, don't! The HP-IB specification says you can turn off 1/3 of the devices connected. But as a general rule, you should always keep all instruments turned on. Loading on the bus from devices you've turned off can cause system errors.

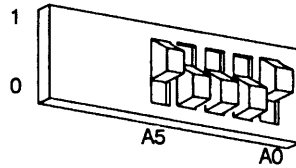
Also, remember the 2-meters-per-device and 20-meters-total cable limits if you start to disconnect instruments.

In the example above, if you disconnect instrument B, the total cable length allowed falls to 4 meters. (2 meters per device.) The cable between controller and instrument A is now too long.

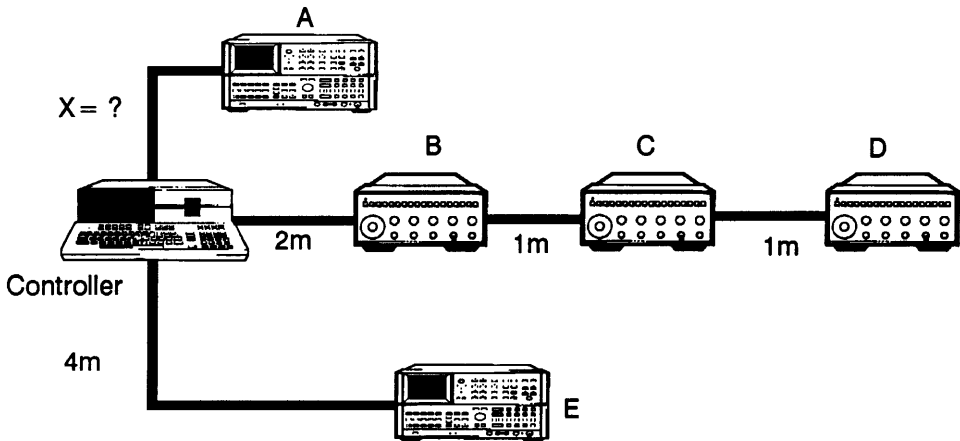
Review Quiz

1. In a dusty attic, you come across a rare model of the Murdstone Modulator. On the instrument's front panel is a decal that reads "with HP-IB" and on the rear is a black HP-IB connector. What capabilities can you safely say the instrument has? Can it respond to any signal on any HP-IB line?
2. The HP 8753A RF Network Analyzer has the following Interface Function Codes:
SH1, AH1, T6, TE0, L4, LE0, SR1, RL1, PP0, DC1, DT0, C0, C1, C10, E2.
What is the capability of this instrument:
 - a. For Acceptor Handshake?
 - b. For Remote Local?
 - c. For Device Trigger?

3. What is the device address of an instrument whose address switch is set as shown here?



4. What is the maximum length of cable X in this configuration?



5. In the previous illustration, what is the maximum length of cable X if you disconnect instrument D?

Take Control of Those Instruments!

In lesson 11, you learned a little about HP-IB and IEEE-488. In lesson 12 you connected your computer to your instruments with HP-IB. Now it's time to use the HP BASIC language to take control of those instruments.

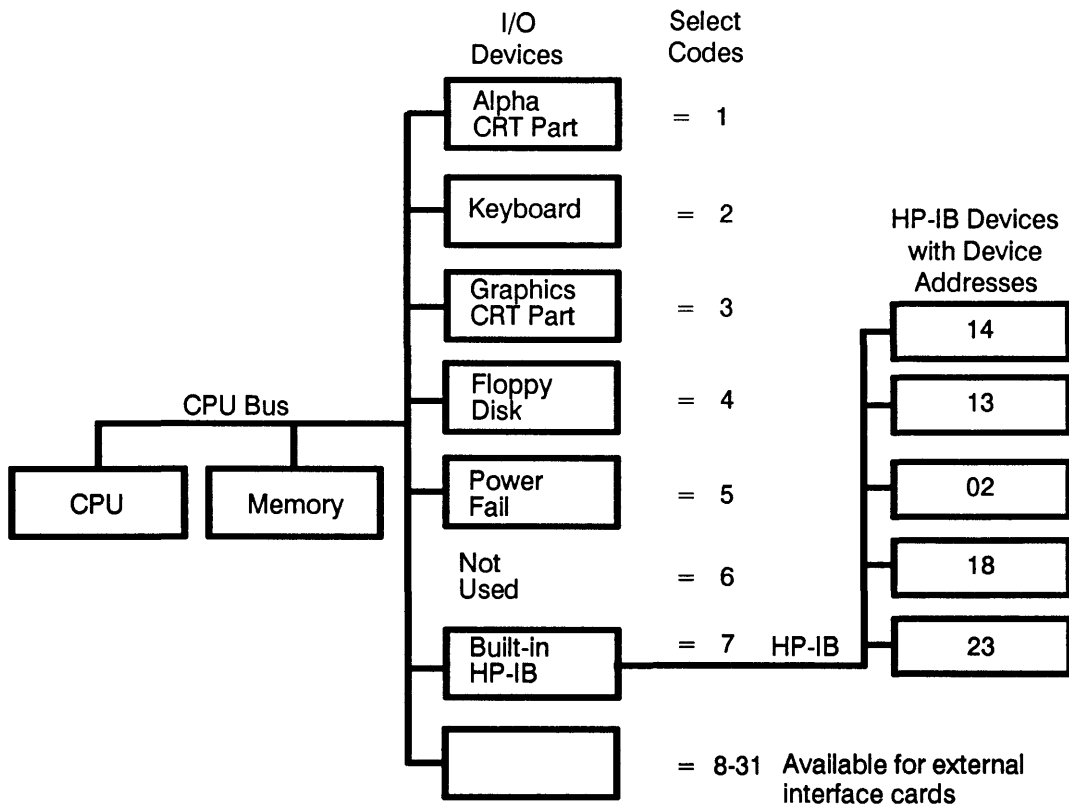
In this lesson you'll learn about:

- ABORT.
- REMOTE.
- The [LOCAL] key and statement.
- LOCAL LOCKOUT.
- CLEAR.

Addressing Instruments (A Quick Review)

Within a computer, the central processing unit (CPU) locates the alphanumeric and graphics portions of the CRT display, the keyboard, and HP-IB by means of select codes. The select code for HP-IB is usually 7.

On HP-IB, each instrument is identified by its address. When the CPU wants to control a specific instrument on HP-IB, it must send a combination of the HP-IB select code and the instrument address, such as "718" or "702".



You don't need to remember the interface select code and instrument address every time you want to address an instrument. Instead, you can refer to it by a name or a path name.

Using a Name

You can use a name for an instrument, just like a variable name. Here's an example:

```
30 Source = 718
```

After line 30 has been executed, you can refer to the instrument at address 718 as "Source," like this:

```
100 REMOTE Source
```

Use names that are easy to remember, such as, oh, "Meter," or "Scope," or "Dvm" (for "digital voltmeter").

Using a Path

You can also use a *path* to the instrument. First, you ASSIGN a path name to that select code and address.

Once you've assigned a path name, from then on you can refer to the device by that easy-to-remember name. For example:

```
10 ASSIGN @Source TO 718
20 CLEAR @Source
```

You can use either an I/O path or a variable name. Using an I/O path (the ASSIGN statement) is faster – it typically gives a 10-20% increase in speed.

Calling Instruments to Attention

If you worked through lesson 11, you remember that the HP-IB was likened to the Pickwick Club, with Pickwick himself as system controller.

The first rule of public speaking at a place like the Pickwick Club is "get the audience's attention." That rule is also true for HP-IB.

Suppose you just walked up to your computer. You know it's the system controller and it's hooked to a few instruments. Moreover, you suspect those sneaky little devils are engaged in some activity—passing data, perhaps, or under control of a different active controller. What *do* you do?

Using ABORT

You take control, that's what! You use the ABORT statement.

ABORT causes all activity on the bus to cease. To try it, type:

```
ABORT 7 ↵
```

Note



The hooked-arrow key signifies an [EXECUTE] (or [ENTER], or [CR]) key on your computer keyboard. Whenever you see this symbol, it means you should press that key.

The ABORT 7 instruction aborts all activity on the interface with select code 7 (usually HP-IB). If you have instruments connected, you may or may not see a response from them. But be assured, they've all stopped what they're doing and are waiting attentively.

Here's what ABORT does:

- Stops all HP-IB activity.
- If the statement is issued by the system controller, it causes the system controller to become the active controller.

Suppose another instrument on the bus is the active controller (while the system controller cools its heels in the background). When sent by the system controller, the ABORT statement transfers control from the active controller back to the system controller again.

The REMOTE Statement

At its simplest, programming an instrument is merely sending it a series of codes that tell it what to do. It's just as if you were standing in front of the device, pressing its buttons and turning its knobs.

When you first turn on an instrument, it "wakes up" in *local* mode. This means that the instrument's front-panel controls take precedence over commands you try to send over HP-IB.

Well, this will *not* do! You want instant, unquestioned obedience. So you send a REMOTE command to the instrument.

Try it on your own instrument. Type REMOTE, followed by your instrument's select code, like this:

```
REMOTE 718_
```

You should see the REMOTE lamp illuminate (or some other indication, depending on the instrument). You'll also see the LISTEN lamp illuminate.

Now you're free to send commands to the instrument over HP-IB. The instrument is in *remote* mode, and all of its front panel controls are disabled.

To put *all* instruments currently addressed to listen on HP-IB in REMOTE mode, you can send the statement to the entire bus, like this:

```
REMOTE 7_
```

This statement is sent *automatically* to all devices whenever:

- The system controller is turned on.
- The system controller sends an ABORT command.
- You execute RESET at the system controller.

Remember, if you don't name a specific instrument, only instruments currently addressed to listen are affected by the REMOTE 7 statement.

Going Back to LOCAL

Now move from the *computer* to the instrument — that is, to the *synthesizer*. If you're sitting in front of an instrument and see its REMOTE lamp come on, you know it's now being controlled via HP-IB. If you turn knobs or push buttons on the instrument itself, they have no effect.

Except for one button: the [LOCAL] key.

The [LOCAL] key on an instrument's front panel puts it back into local mode. It slams the door in HP-IB's face and gives control back to the operator.

Try it. Press the [LOCAL] key on the synthesizer's front panel. (Or on the front panel of your own instrument.) The REMOTE lamp is extinguished.

Now you can change frequency or amplitude from the instrument again. The computer can't affect the instrument until it sends another statement to put it in remote mode again.

Taking Control — And Keeping It

Back at the computer, you see some statements sent over HP-IB aren't affecting the instrument. The operator has pressed the [LOCAL] key.

You have another control trick up your sleeve, though. It's the LOCAL LOCKOUT statement.

LOCAL LOCKOUT disables *all* front-panel control of instruments on the bus. (The instrument must first be in REMOTE mode, though.) A person can't change *any* of the instrument's controls; not even the [LOCAL] button has any effect.

To try it, type:

```
REMOTE 718.␣
```

```
LOCAL LOCKOUT 7.␣
```

Now go to the synthesizer's front panel and press any button, turn any knob. It's all in vain – the computer has absolute control over everything but the power switch. The [LOCAL] button is as ineffective as the other controls. Only the mighty computer can tell the instrument what to do.

The LOCAL Statement

Once the computer has sent a LOCAL LOCKOUT command, there are only three ways an instrument's front panel controls become active again.

1. You turn the instrument power off, then on again.
2. You send a LOCAL statement from the computer (or other active controller).
3. You press the [RESET] key on the computer.

The LOCAL statement cancels REMOTE and LOCAL LOCKOUT. It puts the instruments on the bus back into local mode again. To try it, type:

```
LOCAL 7.␣
```

The REMOTE lamp goes out; you can now control the synthesizer from its front panel again. The LISTEN lamp stays illuminated to show the instrument is still listening to the bus.

Note



You can send a LOCAL statement to a specific instrument (for example, LOCAL 713). However, this *doesn't* cancel LOCAL LOCKOUT.

LOCAL is the only statement that cancels local lockout. (Not even ABORT interferes with the LOCAL LOCKOUT status.)

Using CLEAR

You've already seen how ABORT cancels all HP-IB activity. What if you want to be a little more ... gentle ... in calling instruments to attention? The answer is the CLEAR statement. Try it:

```
CLEAR 718_1
```

This clears only the specified instrument (at HP-IB address 18).

Of course, you can clear *all* instruments on the bus with this statement:

```
20 CLEAR 7
```

This statement re-initializes each device on bus 7 that is capable of responding. (That is, all instruments that are addressed to listen.) CLEAR can only be used by the active controller.

For many HP-IB instruments, CLEAR doesn't affect LOCAL or REMOTE status. It sets the instrument back to its power-on state but *leaves* it in REMOTE (or LOCAL) mode.

This Lesson's Featured Instrument: The HP 3326A Two-Channel Synthesizer

One of the problems in a self-paced course like this one is this:

- Computers are similar.
- HP BASIC is HP BASIC.
- *But:* instruments are different.

This means that you may be happily reading about controlling a power supply and digital voltmeter, but your test setup includes only a function generator and oscilloscope.

For this reason, each lesson will feature one instrument (or perhaps two). You'll learn everything you need to know about the instrument, such as:

- What it does or tests.
- Front-panel controls you'll be programming.
- HP-IB address (the one assigned to it when it's shipped from the factory).
- The necessary command strings that operate the instrument on HP-IB. These are presented just as you'll find them in the instrument's manual.

If you're lucky enough to have that particular instrument hooked up to your computer via HP-IB, you're in luck. You'll be able to follow and try every step in the lesson.

If you don't have the featured instrument, don't despair. These general procedures apply to nearly all HP-IB instruments, and also to many devices compatible with IEEE standard 488-1978.

You can follow along step by step, referring to the instrument description when there's something you don't understand.

In most cases, you'll even be able to try most steps in these pages. Just substitute your own device address and command strings. (You can type all HP BASIC statements, of course, without modification.)

Naturally a *lot* will be different from one instrument to another, including even the front-panel display of such fundamental conditions as local and remote.

However, the statements and programming techniques you learn in this part and part 3 can be used with *all* instruments on HP-IB. Study them carefully and modify them for your own programs.

The HP 3326A Two-Channel Synthesizer

This instrument combines two synthesized signal sources, each of which has a frequency range of from 1 to 13 MHz. It produces sine waves, square waves, dc or pulsed output. You can operate it as:

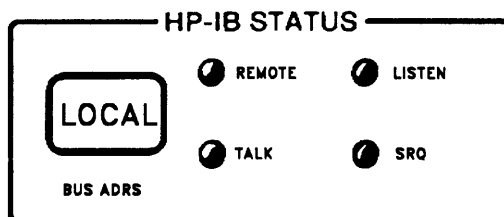
- Two independent signal sources.
- A two-tone source.
- A two-phase source.
- A precision pulse source.

The default synthesizer HP-IB address (as it's shipped from the factory) is 18.

HP-IB functions are: SH1, AH1, T6, L4, SR1, RL1, PP0, DC1, DT1, C0, E1. A quick glance at appendix B of this course tells you the synthesizer has capability to be a listener (L4) or a talker (T6), but not a controller (C0).

In this lesson, we'll only be concerned with one portion of the HP 3326A's front-panel display: the status lights.

These status lights are how the HP 3326A shows its current operating conditions.



A Programmed Example

To see an example program containing many of the statements you've learned about in this lesson, type in the program below. If your instrument is at a different address than the one shown, substitute your instrument's address wherever you need to.

Entering the Program

To enter the program into the computer, follow the procedure given here.

1. Type:

SCRATCH_␣

This clears any program that might already be in the computer.

2. Type:

EDIT_␣

This puts the computer in edit mode, ready for you to enter program statements.

3. Type each program statement exactly as it's shown. Press [EXECUTE] (or [ENTER], or [CR]—whatever is on your computer's keyboard) after each line to "enter" it.

```

10  !REMOTE OPERATION
20  PRINTER is 1
30  Instrument=718
40  REMOTE Instrument
50  PRINT "Instrument is in REMOTE mode"
60  PRINT "Press CONTINUE to see next condition"
70  PAUSE
80  REMOTE Instrument
90  LOCAL LOCKOUT 7
100 CLEAR SCREEN
110 PRINT "Instrument front panel is in LOCAL LOCKOUT"
120 PRINT "Try some front-panel keys"
130 PRINT "Press CONTINUE to see next condition"
140 PAUSE
150 LOCAL 7
160 CLEAR SCREEN
170 PRINT "Local mode. Instrument front panel is active"
180 PRINT "Try some front-panel keys now"
190 END

```

If your instrument is different: If you're using an instrument with a different HP-IB address than 18, change line 30 so the variable "Instrument" is equal to the address of your instrument. That's the only change you'll have to make in the program. If your instrument doesn't have a [LOCAL] key, refer to its manual to determine how to switch from remote to local mode.

Running the Program

To run the program, make sure your instrument is connected and turned on. Then:

1. Press [RUN]. The screen displays:

```

Instrument is in REMOTE mode
Press CONTINUE to see next condition

```

The REMOTE and LISTEN lamps on the instrument should be on. If you try to use any of the instrument's front-panel controls, nothing happens. You may even see an error.

2. There's *one* key you can use, of course. Press the [LOCAL] key on the instrument. The instrument's REMOTE light goes out, and you can operate it from the front panel again.
3. Now go to the computer and press its [CONTINUE] key. The computer displays:

```
Instrument front panel is in LOCAL LOCKOUT
Try some front-panel keys
Press CONTINUE to see next condition
```

When you try front-panel keys now, none of them work. Press [LOCAL]. This key, too, is "locked out" and inoperative.

4. To continue execution, press [CONTINUE] again. The computer screen displays:

```
LOCAL mode. Instrument front panel is active
Try some front-panel keys now
```

The instrument's REMOTE lamp is extinguished. If you press some front-panel keys, you'll see that they're all active again.

Review Quiz

1. Answer both a and b.
 - a. What statement can you use to stop all activity on HP-IB?
 - b. For most HP-IB instruments, what statement can you use to set the instrument at address 7 to its power-on state, without affecting its remote or local status?

2. Which of these, a or b, will probably give faster input and output to the instrument?

a.

```
10 Sweeper = 719
20 REMOTE Sweeper
```

b.

```
10 ASSIGN @Sweeper TO 719
20 REMOTE @Sweeper
```

3. If LOCAL LOCKOUT has *not* been performed, name three ways to change an instrument from remote mode to local.

4. What is wrong with this statement?

```
50 LOCAL LOCKOUT 718,720
```

5. Martin Chuzzlewit has five instruments connected to his computer with HP-IB. They are at the following addresses:

707 712 717 718 725

Chuzzlewit wants to put all these instruments in remote mode. Moreover, he wants the last three (717, 718, 725) to be in a "remote – locked out" condition, so the local operator can *never* affect their operation.

Write a section of HP BASIC code that will do this for him.

Telling Instruments What to Do

At last, you're ready to start controlling instruments! In this lesson you'll learn about:

- Using ASSIGN to open an I/O path.
- How to read a code chart in an instrument manual.
- Using OUTPUT to send instrument codes.
- The all-important terminator.

What the Instrument Needs

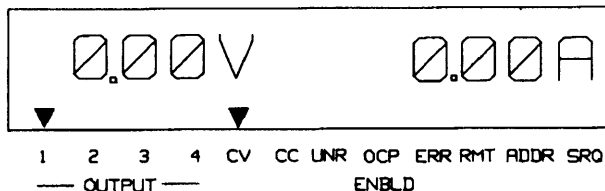
Using HP-IB (or any interface) to tell an instrument what to do is really just another way of operating the instrument. Instead of punching buttons and twisting knobs, though, you use a computer and HP-IB to send the instrument a bunch of characters – ASCII characters – that it recognizes.

When you want to program an instrument to do something, then, it's really a three-step process:

1. Decide how you'd get the instrument to perform the measurement or function manually (that is, in local mode.)
2. Switch the instrument to remote mode.
3. Use OUTPUT to send the necessary ASCII codes to the instrument over HP-IB.

A Manual Example

In this example, you will set a power supply's output voltage to 10 volts. Then you'll set it for an output current limit of 500 milliamperes.



To set the output voltage of channel 1 to 10V, first go to the front panel of the HP 6624A power supply.

Make sure the power supply is in local mode. (You shouldn't see an annunciator arrow over the RMT indicator.) Then:

1. Press the [OUTPUT SELECT] button on the power supply until the annunciator arrow on the display is over OUTPUT 1. (This shows that you've selected channel 1.)
2. To set the voltage of channel 1 to 10 volts, press [VSET] [1] [0] [ENTER]. The power supply's display reads: 10.0V 0.00A.
3. Now set this output to a current limit of 500 mA; press: [ISET] [.] [5] [ENTER]

Now that you've set the output manually, do the same thing with HP BASIC and HP-IB.

Note

The HP 6624A can be set for either constant voltage or constant current operation. In this lesson, you'll use it only in constant voltage mode, with current limiting.

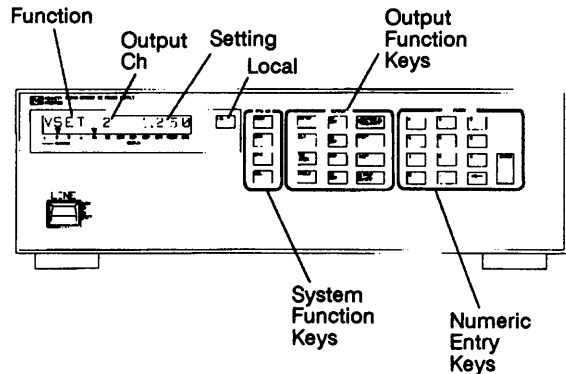
This Lesson's Featured Instrument: The HP 6624A Multiple Output Power Supply

The HP 6624A is one of a family of multiple-output DC power supplies from Hewlett-Packard. It has four separate outputs, and each output has two ranges, as shown here:

Output	Voltage and Current
Output 1	7V @ 5A or 20V @ 2A
Output 2	7V @ 5A or 20V @ 2A
Output 3	20V @ 2A or 50V @ 0.8A
Output 4	20V @ 2A or 50V @ 0.8A

The power supplies are all housed in the same cabinet, and controlled by a single front-panel display. (You switch from one channel to the next with an [OUTPUT SELECT] button.)

You can set overvoltage and current limits individually for each channel.



When shipped from the factory, the power supply's HP-IB address is 5. Naturally, you can change the address if you prefer.

The HP 6624A can be either listener or a talker on HP-IB. The full list of HP-IB interface capabilities is: SH1, AH1, T6, L4, SR1, RL1, PP1, DC1, DT0, C0, E1.

As explained in this lesson, you program the four power supplies by sending instrument command strings (they're actually ASCII strings) using the HP BASIC OUTPUT statement.

Although you can use the same HP BASIC statements for any instrument on HP-IB, the instrument command strings themselves are different for each type of instrument.

Here are a *few* of the instrument commands for the HP 6624A, reprinted directly from the power supply's operating manual.

Command	Description
CLR	Returns the entire power supply (all outputs) to the power on state, except that the supply is not unaddressed and its store/recall registers are not changed.
ISET < ch > , < current >	Sets the current of the specified output channel.
VSET < ch > , < voltage >	Sets the voltage of the specified output channel.

This is just a sample of the instrument commands for the HP 6624A. Its operating manual (like those for other HP instruments) has the full list of commands, along with plenty of examples.

Doing It With HP BASIC

In previous lessons you learned some rudimentary control using statements such as LOCAL and REMOTE, CLEAR, ABORT, etc. These are universal HP BASIC statements, and they have the same effect on all instruments.

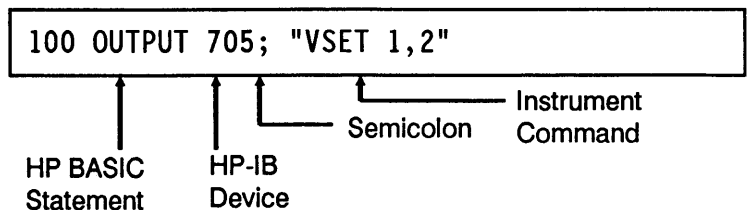
When it comes to actually controlling the *operation* of an instrument, though, things get trickier. You see, instruments are a *lot* different from one another. You'd want to control voltage and current in a DC power supply, for instance; but in a counter, you'd want to be able to change the frequency and amplitude.

For this reason, each HP-IB or IEEE-488 instrument has its own *unique* set of commands. You send the commands to the instrument with the HP BASIC OUTPUT statement.

The OUTPUT Statement

Instrument commands are actually ASCII strings of letters and numbers.

To send a command to an instrument, you normally use the OUTPUT statement. Here's an example:



This means "Output the ASCII string VSET 1,2 to the instrument at HP-IB address 705." The ASCII string VSET 1,2 is the instrument command.

You can use OUTPUT to send instrument commands to:

- A device at an HP-IB address.
- A device referred to by a name.
- A device addressed by an ASSIGNED I/O path.

OUTPUT to a named device: This short section of HP BASIC code shows how you can output an instrument command using a name to refer to a device:

```
100 Pwr_source = 705
110 OUTPUT Pwr_source; "VSET 1,2"
```

Line 100 gives the name Pwr_source to the instrument at HP-IB address 705. Then line 110 outputs the instrument code VSET 1,2 to that instrument.

The advantage of using a name is *unified I/O*: if you change the address of an instrument, you need to change only the line that refers to the address (line 100), not the lines that refer to the instrument by name.

OUTPUT to an I/O path: This section of code shows how you can use ASSIGN to open an I/O path to the instrument, then OUTPUT instrument commands along the path:

```
100 ASSIGN @Source to 705
110 OUTPUT @Source; "VSET 1,2"
```

This is similar to using a label for the instrument. The added advantage is that an I/O path is faster.

For simple commands, it probably doesn't matter; but if you have long command strings or data, an ASSIGNED I/O path can cut execution time.

You can recognize an I/O path by the "at" sign (@) at the beginning of the path name.

Once you've opened an I/O path using ASSIGN, it's good programming practice to *close* the path when you're done. You can do this by ending the program, or by assigning that path to the asterisk (*), like this:

```
110 ASSIGN @Source TO *
```

What OUTPUT Does

In the HP 6624A power supply, as in most HP instruments, the OUTPUT statement from the controller does three things:

1. It puts the instrument in remote mode.
2. It makes the instrument a listener, able to receive commands.
3. It causes the instrument to "listen" for the string data, (and execute the string, if it's a valid command for that instrument).

This means, of course, that when you're using OUTPUT, you often don't need the REMOTE statement, since OUTPUT puts the instrument in remote mode.

Instrument Commands

OUTPUT transfers the instrument command to the instrument just like it transfers data to a mass storage file. (In fact, you may want to review lesson 10 in part 1, for more details of using ASSIGN and OUTPUT.)

The difference, of course, is that the "data" is an instrument command.

At least, it *should* be. Read on.

The key to getting an instrument to do what you want is in its instrument command. And because this command is really just an ASCII string – a bunch of letters and numbers – you have to be extra-careful and vigilant!

Instrument commands, you see, are *unique* to each instrument. And instruments are picky – they don't like it when you send them a command meant for a different device.

You won't always know if you're sending the wrong command, either. Remember, if you try to type an illegal HP BASIC statement, the computer gives you an error right away.

But you can put a statement like this in your program:

```
150 OUTPUT 705; "Yo! Give me 20!"
```

The computer will blithely send this, unaware that there's anything wrong. It may go into its own never-never land while it waits for a response. (A response that – shudder – never comes!) In some cases the instrument may even seem to accept a bad command; but you'll get unpredictable results.

As you may have guessed, there's a cardinal rule afoot:

Cardinal Rule 

Be extra-careful with instrument commands.

Where to Find Instrument Commands

The place to look for instrument commands is in the instrument's manual. These aren't HP BASIC statements; they're the characters that belong *inside quotation marks* after the OUTPUT keyword.

Usually, you'll find instrument commands in an appendix at the back of the instrument's manual.

For your convenience, throughout this course an excerpt from the instrument codes for each featured instrument is contained in that lesson's sidebar. So elsewhere in this lesson, you'll find selected instrument codes for the HP 6624A Power Supply.

Specifying the Instrument Command

Naturally, even if it's a group of difficult-to-remember characters, you can explicitly spell out the string you want to send to an instrument each time, like this:

```
100 OUTPUT 705; "VSET 1,2"
```

However, since the instrument command is really just an ASCII string, you can output it as a variable:

```
100 Voltage$ = "VSET 1,2"  
110 OUTPUT 705; Voltage$
```

Line 100 places the command VSET 1,2 in the string variable Voltage\$.

Then line 110 outputs the contents of the variable to the instrument.

When you output data (whether it's an instrument command string or other data), it's normally sent as a series of 7-bit ASCII characters.

There are times when you'll want to specify the format of data you send to an instrument. You'll learn a simple way to do this in lesson 17, and you'll explore many, many more data formatting options in lesson 25.

An HP-IB Example

Now you know how to set up the power supply manually. And you know how to use the OUTPUT statement, and where to turn to look for the HP 6624A's instrument commands. It's time to put your knowledge to work and *program* the power supply.

Set the Voltage

Here's how you're going to program the power supply. (This assumes, of course, that you've already connected the HP-IB cables from your computer to the instrument.)

- Use the OUTPUT statement.
- Use the instrument's HP-IB address (705 for the HP 6624A).
- Use the instrument command to set voltage.

To find the instrument command, you look in the instrument's manual, and see this:

Command	Description
VSET < ch > , < voltage >	Sets the voltage of the specified output channel.

In this case, the quantities inside the brackets < > are added. So to set the voltage of channel 1 to 10 volts, you type:

```
OUTPUT 705; "VSET 1,10"
```

What about setting the current limit to 500 milliamperes? Again, you look in the instrument manual for what you want to do. Aha, there it is:

Command	Description
ISET < ch > , < current >	Sets the current of the specified output channel.

Thus, to set the output of channel 1 to a current-limit of 500mA, type:

```
OUTPUT 705; "ISET 1, .5" ↵
```

If you look back at how you did this manually, you'll see the programmed instrument commands are astonishingly similar to the power supply's front-panel buttons you pushed.

Vary the Output

Naturally, one of the benefits of programming an instrument such as a power supply is that it can quickly perform tasks that would be extremely tedious if you tried to do them by hand.

Here's an example. This program causes the channel 1 power supply to ramp output voltage in steps of 0.1V from 0 volts to 5 volts. It limits current to 1.0 ampere.

```
10 ASSIGN @Supply TO 705
20 OUTPUT @Supply; "CLR; ISET 1,1"
30 FOR Voltage = 0 TO 5 STEP 0.1
40 OUTPUT @Supply; "VSET 1, "; Voltage
50 WAIT 0.2
60 NEXT Voltage
70 END
```

How it works: Line 10, of course, opens an I/O path to the power supply at HP-IB address 705.

Line 20 returns all power supply outputs to the power-on state; then it sets the current limit of channel 1 to 1.0 ampere.

See the semicolon inside the quotation marks in line 20? This separates the two commands, CLR and ISET 1,1. Line 20 shows how you can link several HP 6624A commands in the same OUTPUT statement.

Note



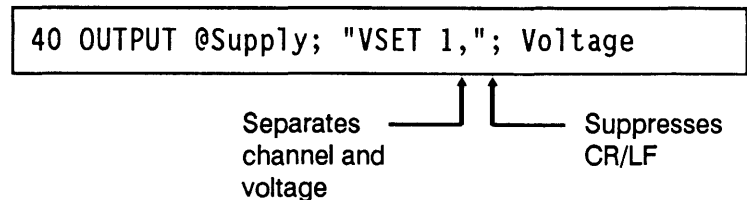
Remember, the instrument commands and structure inside the quotation marks are unique to this instrument. Another instrument may require that you use a comma, or a space, or something else to separate commands inside quotation marks.

Lines 30-60 are a FOR-NEXT loop that increments the voltage in 0.1V steps from 0 to 5 volts.

Line 40 sets the voltage of channel 1 to the variable "Voltage." The comma inside the quotation marks separates the channel number (1) from the voltage (the value of the variable "Voltage"). You could also use a space instead of the comma.

Watch Those End Lines!

In line 40, the semicolon after the quotation marks suppresses the CR/LF (carriage return and line feed) that the computer would send if you used a comma.



If you use a comma instead of a semicolon, the computer sends a CR/LF here, causing an error in the power supply.

You're going to have to watch end-line conditions, and take special care that they occur when needed and are suppressed when they're not.

Instruments are picky! By some estimates, 90% of all instrument lock-ups occur because of CR/LF problems.

A final cardinal rule:

Cardinal Rule 

Watch those end-of-line sequences when programming instruments.

Review Quiz

1. How can you recognize an ASSIGNED path name?
2. Show three ways to output the command "PR" (for preset) to an HP 438A Power Meter at HP-IB address 713.
3. Write a program for controlling the HP 6624A Power Supply that sets channel 1 output to 7.5 volts, and channel 1 current limit to 450 milliamperes.

4. The HP 3314A Function Generator produces sine, square and triangle waves. Its default HP-IB address is 707.

The following program sets the generator to produce a square wave of 1 volt peak-to-peak at 10 kHz:

```
10 OUTPUT 707; "PR"
20 OUTPUT 707; "FU2"
30 OUTPUT 707; "AP1.0V0"
40 OUTPUT 707; "FR 10.0KZ"
50 END
```

Here are a few of the HP 3314A's instrument commands as shown in the generator's manual:

3314A FUNCTION	HP-IB CODES
Amplitude milli-Volt p-p Volt p-p	AP MV VO
Frequency Hertz kilo-Hertz Mega-Hertz	FR HZ KZ MZ
Function off-dc only sine square triangle	FU 0 1 2 3
Preset	PR

Change the program so that:

- a. The output is a sine wave.
- b. The amplitude is 100 millivolts peak-to-peak
- c. The frequency is 1.0 megahertz.

5. Your boss, the Brobdingnagian and bulimic Sampson Brass, has departed for lunch – again. And he’s left you to complete his assignment – again. He’s writing a program to control the HP 6624A Power Supply, but all he’s given you to go on is the scrawled note below.

From the Brass Hat

```
10 ASSIGN @ P3 TO F05  
20 INPUT "Enter a voltage for channel 1", V1  
30 INPUT "Enter a current limit for channel 1", I1  
40
```

Gotta run!

Take over. Prompt the user to enter voltage and current for channel 2, will you? Then output it all to the power supply.

Have the program done before I get back, or else...

Brass

Can you finish the program before Brass returns from lunch?

Getting Information from an Instrument

You're already halfway to complete instrument control. You've learned how to tell an instrument what to do. Now you must get the instrument to give *you* information.

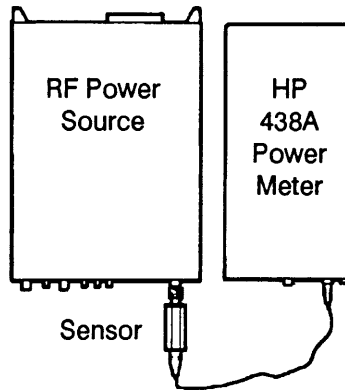
In this lesson, you'll learn about:

- Using ENTER to "read" an instrument.
- Using OUTPUT and ENTER together.
- Triggering an instrument – with or without TRIGGER.
- Using an instrument's internal capabilities.

A Manual Example

To use a voltmeter, you apply a signal to the input and read the voltage on a display. It's the same with many other instruments – counters, oscilloscopes, power meters, etc.

For instance, how do you read power from a power meter such as the HP 438A? You connect the meter to a power source using the correct sensor, and read the power from the instrument's front panel.



Using ENTER

With an instrument connected to a computer via HP-IB, the ENTER statement is almost the same as visually reading an instrument.

Look at this statement:

```
30 ENTER 713;P
```

In the case of the HP 438A Power Meter, the statement means "Read the current power from the meter and place it in the variable P."

If you worked through part 1 of this course, you remember ENTER from lesson 10; you used it to get data from a file on mass storage.

When you "read" an instrument, you're actually getting data from it, too. And with a computer, HP BASIC and HP-IB, you use ENTER to get data from instruments almost like you used it to get data from mass storage.

What ENTER Does

An ENTER statement from the controller to an instrument such as the HP 438A Power Meter does three things:

1. It puts the instrument in remote mode.
2. It makes the instrument a talker, able to send data or status information to the controller (or to another instrument).
3. It assigns data or status information to a variable.

Specifying the Device

Using the ENTER statement, you can specify the instrument or other device in any of three ways:

- With a device address.
- With a device name.
- With an ASSIGNED I/O path.

Using ENTER with a device address: To specify the instrument you're reading, you can use the old, familiar device address, consisting of the HP-IB interface code (HP-IB is usually 7) and the instrument's address.

Here are some examples:

```
ENTER 705;Number
```

Reads the device with address 5 on HP-IB.

```
ENTER 718; Freq
```

Reads the device with address 18 on HP-IB.

Using ENTER with a device name: Another way to specify an instrument is to give it a name, just like a numeric variable. Here's an example:

```
10 Analyzer = 718
20 ENTER Analyzer; Trace_a
```

Line 10 gives the device at address 718 a name of "Analyzer." Then line 20 "reads" the device, putting the current reading in a variable called "Trace_a."

This technique has the advantage of "unified I/O" (you can change the address in line 10, and it automatically changes throughout the program). But input and output of data and commands to and from the instrument is no faster than with a device address.

Using ENTER with an I/O path: You can also use ASSIGN to open an I/O path to an instrument, then use the I/O path with ENTER to "read" the instrument. Here's an example:

```
10 ASSIGN @Pwr_meter to 705
20 ENTER @Pwr_meter; P
```

Line 10 assigns the path @Pwr_meter to the device at HP-IB address 705.

Line 20 then "reads" the device and places the value in variable P. Here's another example:

```
100 ASSIGN @Analyzer TO 718
110 ENTER @Analyzer; Freq
120 ASSIGN @Analyzer TO *
```

In line 100, the path name @Analyzer is opened to the instrument at address 718. Line 110 "reads" the current analyzer value and places it in a variable called Freq.

Line 120 closes the path. Although I/O paths are automatically closed when a program ends, it's a good general practice to close a path (by assigning it to the asterisk, *) if you won't be using it again.

Just as with OUTPUT, using an I/O path with ENTER has two distinct advantages over using a device address or name.

1. An I/O path makes data transfer faster.
2. An I/O path gives "unified I/O": if you change devices or addresses, you need to change only the ASSIGN... TO statement to redirect all parts of the program to the new device.

Specifying the Variable

With OUTPUT, you must send specific strings to the instrument to give it commands. When you read the instrument, though, you're bringing in data and assigning it to a variable.

Since P is a variable, not a string in quotation marks, you can use *any* variable, like this:

```
30 ENTER 713; Power
```

```
30 ENTER 713; P1
```

```
30 ENTER 713; Output_pwr
```

These statements all "read" the meter; they just save the result with different variable names.

You can even take an entire group of readings and place them into an array, like this:

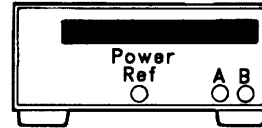
```
270 ENTER 718; Trace_array (*)
```

Of course, all this won't mean much if you don't know *what* you're reading!

This Lesson's Featured Instrument: The HP 438A Dual Sensor Power Meter

The Hewlett-Packard HP 438A is a perfect example of a power meter sized for the field, yet fully capable of computer-controlled testing.

Although it's only 3.5 inches high, the HP 438A has two inputs (channel A and channel B), as well as a 1.0 milliwatt reference oscillator output on the front panel. The LED display shows channel A or B power in watts or dBm. It also reads A – B or B – A; and it shows ratio (A/B or B/A) in percent or dB.



You choose from several sensors to connect to the input jacks. Together these sensors give a frequency range of 100 kHz to 26.5 GHz, over a power range of -70 dBm to +44 dBm (100 pW to 25W).

HP-IB interface compatibility of the 438A is: SH1, AH1, T5,TE0,L4, LE0, SR1, RL1, PP1, DC1, DT1, D0. You can control all instrument functions using HP-IB, with the exception of the power switch, clear entry, and HP-IB address.

Here are *selected* HP-IB programming codes (instrument commands), just as they're shown in the HP 438A's documentation:

Front Panel Accessible

Description	Code
Pre-Measurement	
Preset	PR
Zero	ZE
CAL ADJ	CL
Ref Osc On	OC1
Ref Osc Off	OC0

Front Panel Accessible

Description	Code
Measurement	
Channel A	AP
Channel B	BP
Ratio Measurement	
A/B	AR
B/A	BR
Difference Measurement	
A – B	AD
B – A	BD
Measurement Units	
Log (dBm or db)	LG
Linear (Watts or %)	LN

Non-front Panel Accessible

Description	Code
Trigger Hold Mode	TR0
Trigger Immediate	TR1
Trigger with Delay	TR2
Trigger-Free-Run	TR3
Interface Management	
No Action in Resp to Group Execute Trig	GT0
TR1 in Resp to Group Execute Trig	GT1
TR2 in Resp to Group Execute Trig	GT2

HP 438A Triggering

Before a power measurement can be taken, the meter must be triggered. The 438A has four triggering modes that can be specified under REMOTE control.

TR0: Hold mode. Puts the meter into HOLD mode. No measurements are taken.

TR1: Trigger immediate. Causes the meter to send the next available data point when the 438A is addressed to TALK. The 438A then enters HOLD mode.

TR2: Trigger with delay. Causes the 438A to wait until the digital analog filters have fully settled before taking a reading. The meter then enters HOLD mode, and the reading is held until the 438A is addressed to TALK. TR2 will be used for most REMOTE applications.

TR3: Free run. Puts the meter into free-run mode, continually displaying new readings. The 438A is set to free-run mode upon PRESET.

OUTPUT and ENTER: Hand-in-Hand

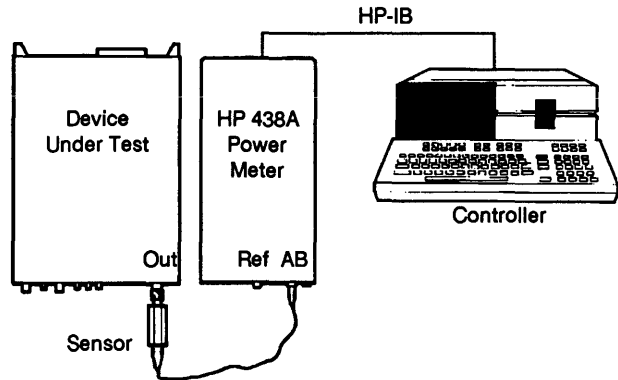
Remember the bad old pre-digital-instrument days? When you read a multimeter, oscilloscope, or power meter, you could see a reading—a meter needle or trace. The question was "what does it mean?"

Chances are, you had to look at a bunch of other buttons and knobs to determine what scale, what range, what value to use.

If you're not careful, instruments on HP-IB can be just as maddeningly obtuse. You know you get data when you use ENTER; the question is, "What data? What does it mean?"

Luckily, OUTPUT comes to the rescue again! You use OUTPUT to "set up" the instrument for the measurement you want. Then you use ENTER to "read" the instrument.

Here's an example of a measurement using the HP 438A Power Meter:



```
10 OUTPUT 713; "PR"  
20 OUTPUT 713; "ZE"  
30 OUTPUT 713; "LN"  
40 OUTPUT 713; "TR2"  
50 ENTER 713; P  
60 DISP P  
70 END
```

Line 10 presets the HP 438A Power Meter. Among other things, this sets it for autoranging.

Line 20 zeros the meter; this takes about 15 seconds. Then line 30 sets the meter to read in watts.

Line 40 triggers the power meter to make a measurement; in this case it's a trigger with delay. It "holds" the reading until you "read" it with ENTER.

All those OUTPUT statements keep the power meter in listen mode. They're all instructions to set it up, then make a measurement.

Now it's time for line 50:

```
50 ENTER 713; P
```

This line puts the power meter in talk mode, then "reads" the value for power (in watts) that was "held" by line 40.

Line 60, of course, causes the computer to display the value for P (that is, the value in watts measured by the power meter). Since you changed it to talk mode with line 50, the HP 438A Power Meter remains in that mode until you change it again.

Here's how OUTPUT and ENTER are used together:

1. OUTPUT makes the measurement.
2. Then ENTER reads the measurement into a variable.

Note



The examples and techniques in this lesson all show data from the instrument as real numbers, unformatted. This means they're read in the computer's internal format, which is usually fine for getting data from most instruments. You *can* get data and output in other formats, too; see lesson 25 in part 3 for more about data formatting.

Triggering an Instrument

When power (or frequency, or amplitude, or a waveform) is constant for a long time, chances are it doesn't matter when you make a measurement.

There are times, though, when you want to know *exactly* when a measurement is made. Or you want several instruments to act at the same time – for instance, a function generator puts out a pulse, and an oscilloscope trace starts simultaneously.

For this, many instruments require a trigger. The trigger says "Make the measurement...*now!*"

The Instrument's Trigger Command

As in line 40 of the example above, sometimes you'll trigger an instrument using the OUTPUT statement and an instrument command string.

The HP 438A Power Meter, for instance, actually has four different triggering modes. You can specify TR0, TR1, TR2, or TR3 with the OUTPUT statement.

Most instruments, including the 438A, also respond to a TRIGGER statement.

The TRIGGER Statement

This is an HP BASIC statement you can use to trigger one device, or several instruments at once. TRIGGER sends a trigger message to a selected device, or to all devices addressed to listen on HP-IB. (For this reason, it's sometimes called a "group execute trigger" or "GET.")

Look at these examples:

```
10 ASSIGN @Hpib TO 7
20 TRIGGER @Hpib
```

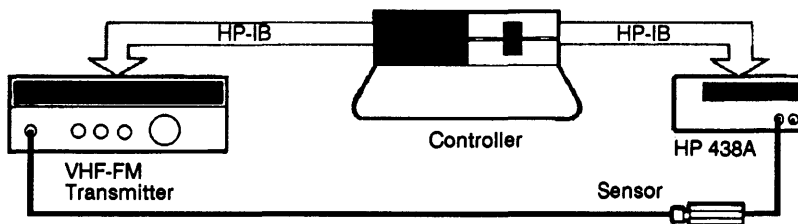
Line 20 above sends a trigger to every device currently in listen mode on the HP-IB.

```
10 ASSIGN @Device TO 707
20 TRIGGER @Device
```

Here, line 20 sends a trigger only to the device at address 07 on the HP-IB interface.

What does TRIGGER do? It depends on the instrument: *You don't know until you look at the instrument's manual.* Many instruments, including the HP 438A Power Meter, let you use instrument commands to control how the device responds to a trigger.

Example: Your assignment is to check the output power of FM-VHF radio transmitters on a production line. The bench setup is simple enough; it looks like this:



The program below uses the HP BASIC TRIGGER statement to take a reading from the HP 438A:

To enter the program, type:

SCRATCH ↵

EDIT ↵

```
10 ABORT 7
20 PRINTER IS 1
30 ASSIGN @Meter TO 713
40 CLEAR @Meter
50 OUTPUT @Meter; "PR AP LN GT2"
60 TRIGGER @Meter
70 ENTER @Meter; Pwr
80 PRINT "The power is"; Pwr; "watts"
90 END
```

Line 50 shows how in the HP 438A multiple instrument commands can be in the same OUTPUT statement. The commands are:

Code	Description
PR	Presets the HP 438A Power Meter.
AP	Sets the meter to measure power on channel A.
LN	Sets measurement units to watts.
GT2	Sets the meter to respond to a group execute trigger as a trigger with delay. (The same as if you had sent OUTPUT "TR2".)

Line 60 is the TRIGGER statement:

```
60 TRIGGER @Meter
```

It triggers everything currently addressed to listen on HP-IB. This includes the 438A, of course, since it was addressed to listen by the previous OUTPUT statement.

You specified "GT2" as the meter's response to a group trigger, so the 438A responds to HP BASIC's TRIGGER statement just as it responds to "TR2". It waits until the analog and digital filters have settled, then takes a reading and holds it until the instrument is addressed to talk.

Line 70 addresses the 438A to talk and reads the value for power into the variable Pwr:

```
70 ENTER @Meter; Pwr
```

To run the program:

1. Make sure all hardware is connected, including the sensor between the HP 438A and the transmitter.
2. Key the transmitter.
3. Press RUN on the computer.

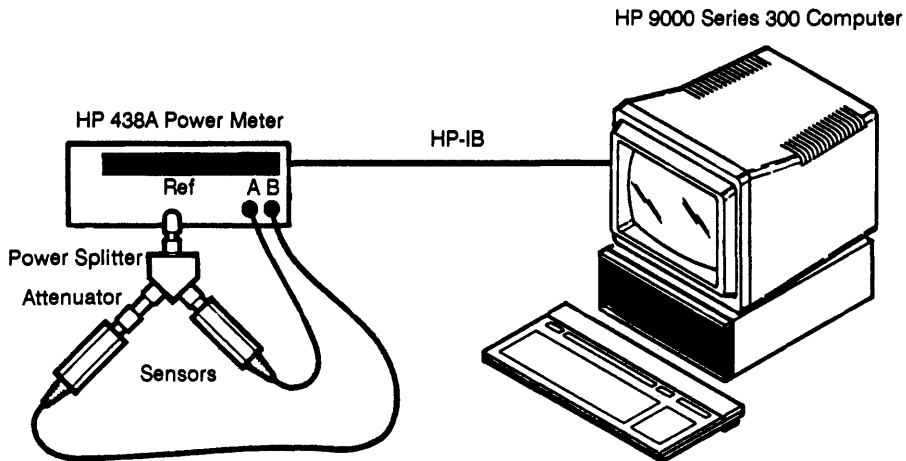
The computer display shows something like this:

The power is 2.23 watts

Instruments Are Smart Too!

When you write instrument control programs, don't forget to use the innate intelligence of the instruments themselves. Often you can save much time and many programming steps by exploiting the full range of instrument features.

Example: Never the most original of thinkers, Thomas Gradgrind is trying to test attenuators using an HP-IB-controlled 438A Power Meter.



Gradgrind is smart enough to use the HP 438A's internal reference oscillator and dual-input capability, along with a power splitter.

He plans to feed a known output from the oscillator through a splitter. Half the power will pass through an attenuator and half will not. Then he plans to use the channel A power and channel B power to calculate the ratio of A to B, and the difference (A – B), which together will tell him what he wants to know about the attenuator.

Luckily for Gradgrind, there's less calculation involved than he thinks. That's because the HP 438A has functions built in to compute A/B and A – B.

To see the easy way to check Gradgrind's attenuator, look at the program.

```
10 ABORT 7
20 ASSIGN @Meter TO 713
30 CLEAR @Meter
40 OUTPUT @Meter; "PR OC1 LG TR2"
50 ENTER @Meter; A
60 PRINT CHR$(12)
70 PRINT "CHANNEL A POWER IS "; A; "watts"
80 OUTPUT @Meter; "BP TR2"
90 ENTER @Meter; B
100 PRINT "CHANNEL B POWER IS "; B; "watts"
110 OUTPUT @Meter; "AR TR2"
120 ENTER @Meter; Ar
130 PRINT "THE RATIO A/B IS "; Ar;"dB"
140 OUTPUT @Meter; "AD TR2"
150 ENTER @Meter; Ad
160 PRINT "THE DIFFERENCE A-B IS "; Ad; "watts"
170 END
```

How it works: The OUTPUT statement in line 40 presets the HP 438A Power Meter, turns on the reference oscillator, selects log mode, and triggers a measurement. Line 50 reads what the meter measured on channel A, and stores the value in variable A.

Line 60 prints a form feed, advancing the paper (or CRT screen) to the top of the next page. Then line 70 prints the value for channel A power.

The statements in lines 80-100 change the 438A to read channel B power, then print the value.

The OUTPUT statement in line 110 sets the meter for A/B ratio calculation, then triggers the measurement. The ENTER statement in line 120 reads this value, and line 130 prints it.

The difference of $A - B$ is also calculated internally by the 438A, then printed.

Of course you could have written a program to actually calculate A/B and $A - B$. But it's easier to use the 438A internal functions.

There's a cardinal rule lurking here:

Cardinal Rule 

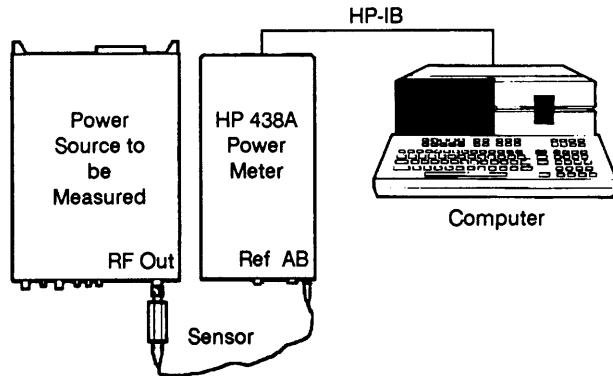
To make your programs simpler, let your instrument do the work.

Review Quiz

1. What HP BASIC statement can you use to "read" any instrument and place the result into a variable in the computer?
2. What are the two ways of triggering a measurement on the HP 438A Power Meter (and most instruments)?
3. Rewrite this program segment for unified I/O – that is, use an I/O path to HP-IB and to the instrument (it's a spectrum analyzer) at address 718.

```
10 ABORT 7
20 CLEAR 718
30 OUTPUT 718; "MKA?"
40 ENTER 718; Amp_marker
50 OUTPUT 718; "MKF?"
60 ENTER 718; Freq_marker
70 PRINT Amp_marker, Freq_marker
80 END
```

4. A HP 438A Power Meter is set up for measurement as shown here:



Write a brief program that will

- a. Abort all activity on HP-IB.
- b. Clear the 438A power meter (at address 713).
- c. Preset the meter.
- d. Zero the meter.
- e. Set the meter to measure watts.
- f. Set the meter for channel B.
- g. Use trigger immediate (TR1) to trigger the meter.
- h. Read the value into a variable called "Watts."
- i. Print the value of "Watts" on the screen.

How an Instrument Summons Service

Ponder for a moment the poor instrument: when the controller sends a command string with **OUTPUT**, the instrument does as it's told. When the controller executes an **ENTER** statement, the device delivers data.

But what if the instrument is busy, and can't process the **OUTPUT** or **ENTER** statement? Or even – heaven forbid – the device should want to request service itself? (Perhaps because the device has finished with its measurement, or because an error has occurred.) What then?

This lesson shows how instruments can be given a "voice" on HP-IB. In this lesson you'll learn about:

- Using **STATUS** to read the HP-IB status register.
- Using **BIT** to determine a bit's setting.
- Using **ENABLE** and **ON INTR** to interrupt a program.
- Using **SPOLL** or commands to read an instrument's status byte.

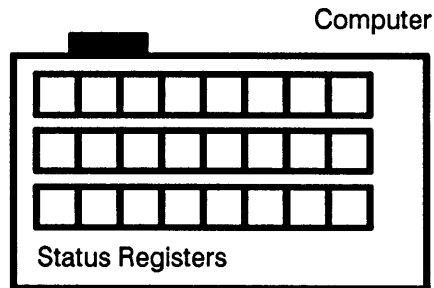
This lesson shows how an instrument can get service – and respect – from the computer.

Requesting Service

When an instrument wants service – because of an error, say, or because it’s ready to accept a command string – it can send a service request (SRQ) out on HP-IB.

The service request doesn’t cause the computer to take any action. In fact, all the SRQ does is to turn on specific bits in the computer’s HP-IB status registers, as well as turn on one bit within the instrument – in its status byte.

All interfaces have status and control *registers* associated with them. These registers are actually memory locations on interface cards or within the computer. They’re used by the computer to keep track of what’s happening.



There are many kinds of status and control registers. One set of registers maintains information on I/O paths, while another tells the current CRT status (what color, what the next print position is, etc.). There are also registers for the keyboard, as well as a set of registers for HP-IB status and control.

How big is a "register"? It depends. Some registers are just a few bits; others have one or even two 8-bit bytes of information.

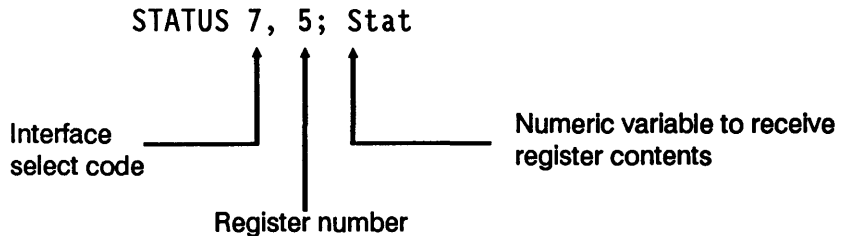
Here's the difference between *status* and *control* registers:

- Status registers let you read status but not change it.
- Control registers let you actually go in and change bits to control the status of an interface.

In this part of the course, you won't be directly changing any control registers – you'll just read status registers.

Reading Registers with STATUS

To "read" a status register, you use the STATUS statement, along with the interface select code, the register number (optional), and a numeric variable to receive the register contents, like this:



This statement reads the value of status register 5 on interface 7. (The interface with select code 7 is usually HP-IB.) The value of that register is placed in a variable called Stat.

If you don't specify a register number, you'll get the status of register 0 of the interface.

You can use STATUS to read more than one register in a row; the "read" begins with the register number you specify, and continues for the number of variables you have.

Example: Unless you've changed it, the CRT display is at select code 1. Here are a couple of CRT status registers, as they're shown in the *BASIC Condensed Reference*:

STATUS Register 12 Key labels display mode:
 0 = typing-aid key labels displayed.
 1 = key labels always off.
 2 = key labels always on.

STATUS Register 13 CRT height (number of lines to be
 used for alpha display.)

Type this mini-program:

SCRATCH_↵
EDIT_↵

```
10 PRINTER IS 1
20 STATUS 1, 12; Keymode, Height
30 PRINT "Key mode is"; Keymode
40 PRINT "CRT height is"; Height; "lines"
50 END
```

Now switch out of edit mode; press:

[PAUSE]

Then type:

KEY LABELS ON_↵
RUN_↵

The screen displays something like this:

```
Key mode is 2
CRT height is 25 lines
```


Now turn off the softkey labels at the bottom of the screen and try again. Type:

```
KEY LABELS OFF ↵  
RUN ↵
```

The screen displays:

```
Key mode is 1  
CRT height is 25 lines
```

How it works: Line 20 "reads" two status registers of interface 1, beginning with register 12. It places the contents of register 12 in the variable Keymode, and the contents of register 13 in the variable Height.

When you use the KEY LABELS ON or KEY LABELS OFF statement, it changes the contents of status register 12, yielding a different output from your mini-program.

HP-IB Status Registers

The HP-IB has eight status registers (and eight control registers). You'll find a complete list of them in appendix C.

Most of the HP-IB status registers contain two 8-bit bytes of information.

Register 4, for instance, has two bytes (16 bits) of information. The bits are labeled 0-15, and each bit is "binary-weighted." So if bit 1 (SRQ received) is set to 1, and all other bits are off (i.e., 0), the statement:

```
100 STATUS 7, 4; Stat
```

puts the value 2 in the variable Stat. (Because the decimal value of bit 1 is 2.)

Each set of HP-IB status and control registers has a unique function. Register 4, for example, tells what condition *caused* an interrupt, while register 5 (a mirror image of register 4) tells what interrupts are *currently enabled*. (That is, what interrupts are currently allowed to occur.)

STATUS Register 4

Interrupt Status

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

Register 7 shows the current status of the HP-IB bus control and data lines.

STATUS Register 7

Bus Control and Data Lines

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
ATN True	DAV True	NDAC True	NRFD True	EOI True	SRQ True	IFC True	REN True
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
DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

When an SRQ Is Issued

When an instrument requests service by issuing a service request (commonly called an "SRQ"), it makes the HP-IB SRQ line true (set to 1).

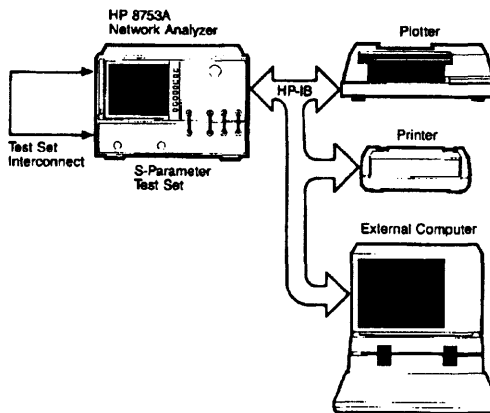
If you look at HP-IB status registers 4 and 7, you can see that *both* of them respond to an SRQ:

- Bit 1 of status register 4 is set to 1. (If enabled.)
- Bit 10 of status register 7 is set to 1. (If this is the active controller.)

Once any instrument on the HP-IB issues a service request the SRQ line (and these bits) stay *true* until you reset them. So even if there's a momentary glitch in one instrument, the SRQ line stays true until you can find out *which* instrument and *what* glitch.

This Lesson's Featured Instrument: The HP 8753 Network Analyzer

The HP 8753 is an easy-to-use measurement system with a large, annotated display, its own set of softkeys, and two independent channels for measuring transmission and reflection characteristics simultaneously. The analyzer covers a frequency range of 300 kHz to 6 GHz, with resolution to 1 Hz.



Data can be displayed on the analyzer's CRT screen in many different formats: log magnitude, linear magnitude, SWR, phase, group delay, polar, real, or Smith chart. The two independent display channels can be viewed separately or at the same time.

HP-IB Capability

All HP 8753 functions are completely programmable. The instrument's default HP-IB address is factory set at 16 (it can easily be changed), and its capability codes are: SH1, AH1, T6, TE0, L4, LE0, SR1, RL1, PP0, DC1, DT0, C0, C1, C10, E2.

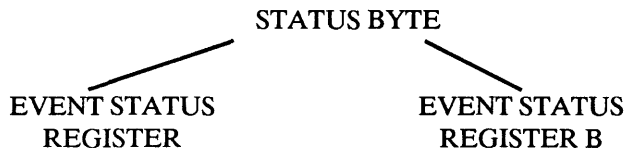
Under HP-IB control, the HP 8753 can operate as a talker/listener, as a system controller, or as a pass control device. Because the 8753 can act as the active controller, you can store or print data or print on HP-IB devices without using a computer.

Here are the HP 8753 instrument command strings you'll use in this lesson:

ACTION	MNEMONIC	DESCRIPTION
Clear	CLES	Clears the status byte.
Interrogate	ESB?	Returns event status register B.
	ESR?	Returns the event status register.
	OUTPSTAT?F255D	Returns the status byte.
Enable	ESE[D]	Enables event status register. (0 < D < 255)
	ESNB[D]	Enables event status register B. (0 < D < 255)
	SRE[D]	Enables SRQ. (0 < D < 255)
Error	OUTPERRO	Outputs the oldest error in the error queue. The error number is transmitted, then the error message, in ASCII format.
Status byte	OUTPSTAT	Outputs the status byte. ASCII format.

Status Reporting

The HP8753A status reporting structure consists of three registers:



The top level register is the status byte, which consists of summary bits. Each bit reflects the condition of another register or a queue. If a summary bit is set (equals 1), the corresponding register or queue should be read to obtain the status information and to clear the condition.

Reading the status byte, which can be done with a serial poll or by issuing **OUTPSTAT**, does not affect the state of the summary bits: they always reflect the condition of the summarized queue or register. Any bit in the status byte can be selectively enabled to generate a service request (SRQ) when set.

Setting a bit in the service request register with **SREnn** enables the corresponding bit in the status byte. For example, **SRE24** enables status byte bits 3 and 4 (since $2^3 + 2^4 = 24$), and disables all the other bits. **SRE** will not affect the state of the status register bits.

The event status register and event status register B are the other two registers in the status reporting structure. They are selectively summarized by bits in the status byte via enable registers.

The event status registers consist of latched bits. A latched bit is set at the onset of a specific trigger condition in the instrument, and is cleared only by a read of the register. The bit will not be set again until the condition occurs again. If a bit in one of these two registers is enabled, it is summarized by the summary bit in the status byte. The registers are enabled by **ESEnn** and **ESNBnn**, which work the same as **SREnn**.

If a bit in one of the event status registers is enabled and the summary bit in the status byte is enabled, an SRQ will be generated when the event status register bit is set. The SRQ will not be cleared until one of four things happens:

1. The event status register is read, clearing the latched bit.
2. The summary bit in the status byte is enabled.
3. The event status register bit is disabled.

or

4. The status registers are cleared with CLES; or a preset.

SRQ's generated when there are error messages or when the instrument is waiting for Group Execute Trigger (GET) are cleared by reading the errors or issuing GET, disabling the bits, or by clearing the status registers.

Error Output

When an error condition is detected in the HP 8753, a message is displayed on the instrument, and that message is placed in an error queue within the analyzer. The error queue holds up to 20 errors (in the order they occur) until you read them out using the OUTPERRO command.

If there are any errors in the queue, bit 3 of the instrument's status byte is set.

The instrument command OUTPERRO causes the instrument to output one error message, which consists of an error *number* followed by an ASCII *string* of up to 50 characters.

How to Detect the SRQ

Remember, even though the SRQ line is true and the SRQ bits in the computer's HP-IB status registers are set to 1, the computer doesn't automatically take action. No indeed!

You have to *detect* the SRQ. There are several ways to do this; here are a couple:

1. Use STATUS to read the SRQ bit on one of the status registers.
2. Use ENABLE INTR and ON INTR to allow interrupt status register 4 to interrupt a program.

Detecting a Status Change

One way of detecting a service request is to use STATUS to "look at" one of the SRQ bits in the status registers.

Here's an example:

```
100 LOOP
110 STATUS 7,7; Stat
120 IF BIT (Stat, 10) = 1 THEN GOSUB Service
130 END LOOP
```

In this example, line 110 "reads" the value of HP-IB status register 7 into the variable called Stat.

In line 120, BIT looks at the entire byte, and returns a 1 or a 0 representing the value of the specified bit of its argument.

What BIT (Stat, 10) does is:

1. Converts the value in Stat from a decimal value to a 16-bit binary byte.
2. Looks at bit 10 of the byte to see if it's a 1 or a 0.

Thus line 120 of the program branches to the subroutine "Service" if bit 10 (the SRQ line) of HP-IB status register 7 is true.

Incidentally, you don't need to use the "= 1"; the BIT statement assumes you mean "= 1" every time. So these two statements are the same:

```
10 IF BIT (Stat,2) = 1 THEN GOSUB Handle
```

```
20 IF BIT (Stat,2) THEN GOSUB Handle
```

To specify "= 0", though, you have to be specific:

```
130 IF BIT (Stat,4) = 0 THEN 120
```

Example: "Lines, shmimes!" snorts the suspicious Harold Skimpole. "Registers, smegisters! I don't trust anything I can't see."

Specifically, Skimpole wants to see a readout of what the HP-IB lines are set to at any one time.

You can help him. Your disk of examples has a handy program to help analyze the current state of HP-IB. To load the program "HPIB_LINES" and list it, type:

```
LOAD "HPIB_LINES" ↵  
LIST ↵
```

```

10 !RE-STORE "HPIB_LINES"
20 PRINTER is 1
30 PRINT
40 PRINT "HP-IB Interface Status"
50 PRINT
60 STATUS 7,7; Bus
70 PRINT "Bus line decimal total is"; Bus
80 IF BIT (Bus,8) THEN PRINT "REN true"
90 IF BIT (Bus,9) THEN PRINT "IFC true"
100 IF BIT (Bus,10) THEN PRINT "SRQ true"
110 IF BIT (Bus,11) THEN PRINT "EOI true"
120 IF BIT (Bus,12) THEN PRINT "NRFD true"
130 IF BIT (Bus,13) THEN PRINT "NDAC true"
140 IF BIT (Bus,14) THEN PRINT "DAV true"
150 IF BIT (Bus,15) THEN PRINT "ATN true"
160 END

```

The program checks each of bits 8-15 of HP-IB status register 7 (assuming HP-IB is at address 7) and tells you if that line is on.

To run the program, just type:

RUN_

The display will vary, depending on the state of HP-IB in your computer. Here's a typical one:

```

Bus line decimal total is-24225.
REN true
NDAC true
ATN true

```

Besides soothing Skimpole, you can use this program whenever you want to find out the current state of HP-IB.

This program also shows how you can modify program execution based on the setting of *any* bit (not just SRQ).

Using an Interrupt

If you look closely at using STATUS to "read" a status register, you'll see that the "read" is usually done from within a loop in the program. And while this is OK for suspending execution temporarily (if you're waiting for the instrument to finish making a measurement, for instance), it's not the best way to handle errors and some other conditions.

Sometimes you'll want to use an *interrupt* to request service. An interrupt is something that lets a program continue. It doesn't need a loop.

Example: The program "8753_INTR" on your disk of examples shows an interrupt that occurs if you send the HP 8753 Network Analyzer an illegal instrument command string. Here's the code:

```
10  !RE-STORE "8753_INTR"
20  ABORT 7
30  ASSIGN @Inst TO 716
40  REMOTE @Inst
50  ON INTR 7 GOSUB Message
60  ENABLE INTR 7;2
70  OUTPUT @Inst; "CLES;" !Clears 8753 status bytes
80  OUTPUT @Inst; "ESE 32;" !Enables bit 5 of 8753 event status register
90  OUTPUT @Inst; "SRE 32;" !Enables bit 5 of 8753 status byte
100 FOR I=1 TO 100
110 WAIT .5
120 PRINT I
130 NEXT I
140 STOP
150 Message: !
160 PRINT "You sent me a BAD instrument command!"
170 OUTPUT @Inst; "ESR?"
180 ENTER @Inst; Estat
190 PRINT "My status byte now contains";Estat
200 !ENABLE INTR 7
210 RETURN
220 END
```

Remember, the instrument command strings and the separators – everything within quotation marks after the OUTPUT statement – depend on the instrument you're programming. The HP 8753 and many other instruments use semicolons or commas as separators, but your instrument may be different.

When you run this program (you'll have to have an HP 8753 connected, or change the program for your own instrument), the computer begins printing the value of I each time through the loop:

RUN_␣

```
1  
2  
3
```

The loop is executed with nothing else happening – until you use the "live" keyboard to OUTPUT a bad instrument command string to the HP 8753. At the computer keyboard, type:

```
OUTPUT @Inst; "HELLO"
```

The computer immediately interrupts its execution of the loop and prints:

```
You sent me a BAD instrument command!  
My status byte now contains 32  
11  
12  
13
```

Then execution continues with the loop.

How it works: The ON INTR 7 statement in line 50 isn't inside the loop. But it "remembers" that if HP-IB (address 7) is interrupted, execution branches to the "Message" label.

Another statement goes hand-in-hand with ON INTR: it's the ENABLE INTR statement in line 60. ENABLE INTR determines what conditions are allowed to interrupt execution.

The conditions on HP-IB that cause an interrupt are found in HP-IB status register 4, the Interrupt Status Register. These conditions are enabled by register 5, the Interrupt Enable Mask. Register 5 contains the bits you specify with ENABLE INTR, and "allows" only those bits to turn on register 4.

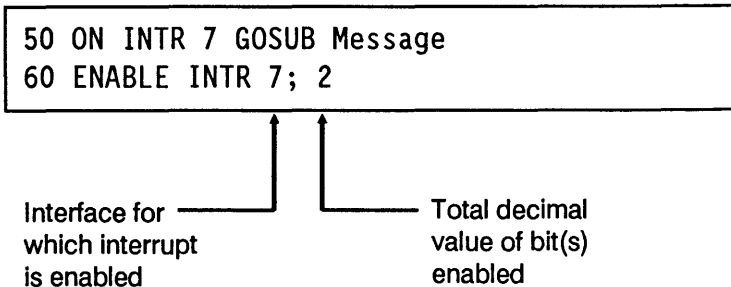
STATUS Register 5

Interrupt Enable Mask

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

You can see that line 60 of the program enables only bit 1 (decimal value 2). When this bit is set to a 1, it indicates there's been an SRQ issued somewhere on HP-IB.



Line 50 enables program interruption according to the setting specified in line 60. So it interrupts execution only if bit 1 (decimal value 2) of the computer's interrupt enable mask (register 5) is set. In essence the computer says "The only interrupt I recognize – from anybody on the bus – is bit 1 of my status register."

When bit 1 *is* set, line 50 branches execution to a subroutine called "Message."

The way the program is written now, the interrupt works only once. Why? Because line 200 is "commented out."

```
200 !ENABLE INTR 7
```

ENABLE INTR, you see, is a "one-shot" statement: When the enabled interrupt finally occurs, it has to be re-enabled if you want to use it again. To re-enable the interrupt, you can remove the exclamation point from line 200 and turn it into another executable ENABLE INTR statement.

```
200 ENABLE INTR 7
```

Cardinal Rule

Be sure to re-enable an interrupt after you process it.

If you don't specify a bit mask value in `ENABLE INTR`, it assumes the value in the previous `ENABLE INTR` statement is still active. That's why *in this program* these two statements are equivalent:

```
200 ENABLE INTR 7; 2
200 ENABLE INTR 7
```

Enabling Other Interrupts

The interrupt you'll use most often is the SRQ. But you can use other interrupts as well. For instance, bit 6 of the computer's HP-IB register 5 enables an interrupt if a device doesn't respond to the interlocking handshake during an `OUTPUT` statement. (This could indicate a device is not connected or its power is off). To enable the interrupt, you use an `ENABLE INTR` statement to "turn on" this bit.

Here's an example:

```
500 ENABLE INTR 7;64
600 ON INTR 7 GOSUB Process
```

Notice that it doesn't matter whether the `ENABLE INTR` statement is *before* or *after* the `ON INTR` statement.

If you don't specify a bit mask value, and there was no prior `ENABLE INTR` statement, the value of 0 is used. This *disables* all interrupts, so the program won't be interrupted.

If you have an instrument—any instrument—connected to your computer with HP-IB, you can rewrite the program to interrupt on an error condition in your own instrument. Look at your instrument's manual to find out more details of how it can generate interrupts.

Do You Need to Mask the SRQ?

One more thing about interrupts. Some instruments such as the HP 8753 "wake up" unable to generate an SRQ. That's because their *status byte* inside the instrument is masked.

And what's a status byte? Glad you asked.

The Status Byte Tells All

Suppose you have many instruments on HP-IB. How does the computer determine *which* instrument requested service?

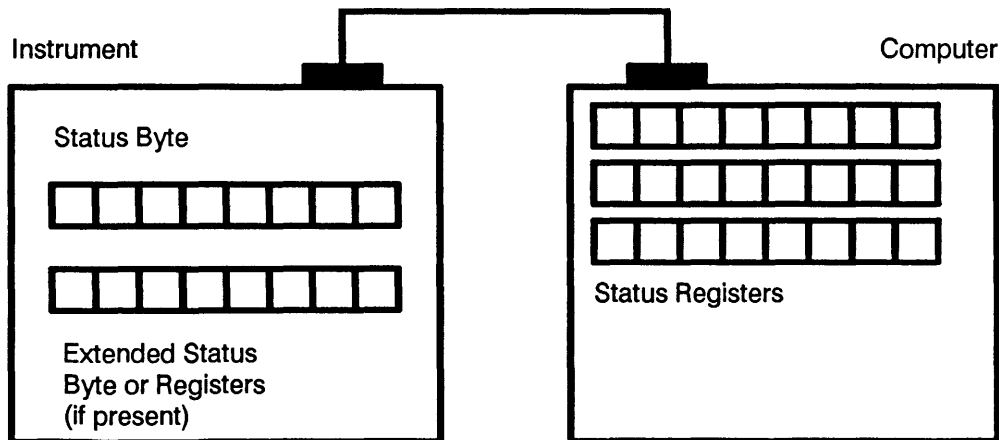
Or what about instrument conditions? How can the computer find out about instrument-specific errors and other status?

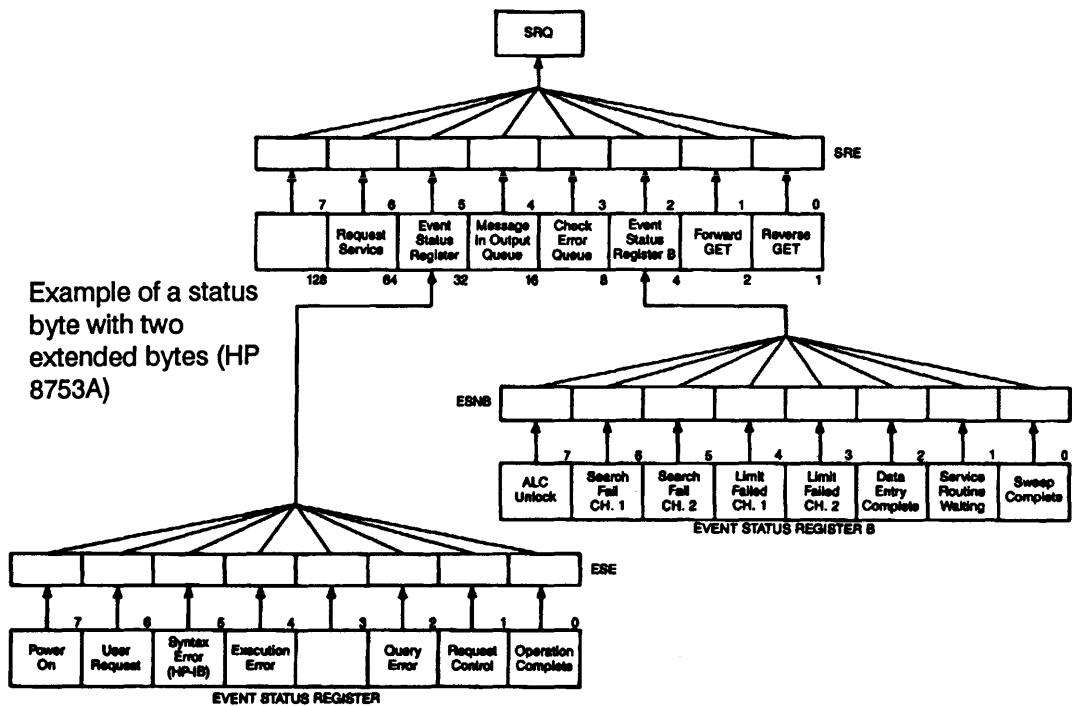
That information is in the instrument's *status byte*.

The Status Byte

Instruments *are* different. But all HP-IB instruments have a status byte. Don't confuse HP-IB status registers and instrument status bytes.

- Status *registers* are in the HP-IB interface in the computer.
- Status *bytes* are within individual instruments.





Example of one-byte status byte (HP 6030 Power Supply)

Condition	* RQS	ERR	RDY	* *	* PON	FAU		
Bit Position	7	6	5	4	3	2	1	0
Bit Weight	128	64	32	16	8	4	2	1

Where * - Not Used
 FAU - Fault Condition
 PON - Power on Reset
 RDY - Ready to Process Commands
 ERR - Programming Error
 RQS - Requesting Service

Here's the difference:

- The HP-IB status registers tell you about the state of the HP-IB interface.
- An instrument's status byte gives you specific information about that instrument.

The status byte of one instrument is different from another. Some instruments, like the HP 6030 Power Supply, have only a single byte; others, such as the HP 8753 RF Network Analyzer, have additional bytes. (Here they're called "registers.") You get to the information in these "extended" bytes in different ways, depending on the instrument.

In the case of the HP 8753, if any of the bits in its event status register is set to 1, it also sets bit 5 of the status byte and sends an SRQ. If any of the bits in event status register B is set, it sets bit 2 of the instrument's status byte. Bits 2 and 5 are called "summary bits", because they summarize the condition of the extended registers.

The status byte is sometimes called the *status word*, or, as in the HP 6030 Power Supply, the *serial poll register*. Except for bit 6 – which on HP-IB is always used to request service – you'll have to look in the instrument's manual to find out what's in its status byte.

Unmasking the Status Byte

Remember the program "8753_INTR" that showed how to interrupt a program? When the HP 8753 wakes up, its status byte is masked – it doesn't generate SRQ's or give you any other information.

When an instrument awakens this way, you have to unmask the status byte, using an instrument command for the purpose. In the case of the HP 8753, you use this code:

```
70  OUTPUT @Inst; "CLES;" !Clears 8753 status bytes
80  OUTPUT @Inst; "ESE 32;" !Enables bit 5 of 8753 event status register
90  OUTPUT @Inst; "SRE 32;" !Enables bit 5 of 8753 status byte
```

Line 70 of this sequence clears the instrument's status reporting system. Then line 80 enables bit 5 of the event status register, allowing a syntax error to be reported. Line 90 enables bit 5 of the status byte, which allows conditions reported by the event status register to generate an SRQ.

Not all instruments awaken with their status byte masked. In fact, you'll even see instruments with an [SRQ] key right on the front panel, so you can generate an interrupt with the press of a button.

The Beauty of Bit 6

In IEEE-488 instruments, bit 6 of the status byte is always the bit that is set to request service.

Remember, the status byte for each instrument is different – *except* for bit 6 of the instrument's main status byte.

Notice that in both the HP 8753 and the HP 6030 Power Supply, bit 6 of the status byte is the same: it is used to request service from the controller. By the definition of IEEE-488, an instrument that requests service will have bit 6 of its status byte true (set to a 1).

Cardinal Rule

Instruments and their status bytes are different. But bit 6 of every HP-IB instrument's status byte is always set by that instrument's SRQ.

Reading the Status Byte

You *could* use STATUS to "read" an instrument's status byte, like this:

```
110 STATUS 716; A
```

This puts the value of the status byte in the variable A.

But STATUS puts the instrument into remote mode to read its status byte. A much better statement for reading a status byte is SPOLL.

SPOLL (serial poll) goes to each instrument on the bus (starting with the lowest-numbered address) and looks at its status byte.

Try this: with your own instrument connected (use its address), type:

```
PRINT SPOLL(716), (Use your instrument's address.)
```

You'll see the current status of your instrument's status byte.

Here are some examples of SPOLL statements:

```
10 Stat= SPOLL (707)
```

This reads the value of instrument 707's status byte into a variable called Stat.

```
100 ASSIGN @Device TO 716  
110 Status = SPOLL (@Device)
```

Line 110 uses an assigned I/O path to read the value of the status byte of the instrument at address 716.

SPOLL (serial poll) is an HP-IB function dedicated to getting status information quickly. It doesn't cause the instrument to go into remote mode.

This means you can use SPOLL to find out (and change) an instrument's status even if it's being operated from the front panel. For instance, you could permit a user to operate the instrument until he or she committed an error, then "lock out" the front panel and return control to the computer.

You can use SPOLL to find out *which* instrument wants service. Here's an example of how to poll two devices:

```
100 LOOP
110 STATUS 7,7; Stat
120 IF BIT (Stat,10)= 1 THEN GOSUB Service
130 END LOOP
140 !

480 Service:!
490 Source_stat= SPOLL (717)
500 Volt_status= SPOLL (722)
510 IF BIT (Source_stat,1) THEN GOSUB Treat
520 IF BIT (Volt_status,1) THEN GOSUB Read_data
530 RETURN
540 Treat:!
    Code for Treat subroutine here
850 RETURN
860 Read_data:!
    Code for Read_data subroutine here
950 RETURN
960 END
```

In the main program loop (lines 100-130), when the SRQ line goes true, the program branches to the subroutine "Service." It does a serial poll on each instrument, then branches further to take care of any service requests.

When the computer determines that a particular instrument is calling for service, it can then analyze the instrument's status byte (or bytes) to find out just *why* the device wants service.

In the program segment above, line 500 assigns the value of instrument 722's status byte to the variable "Volt_status." Then line 520 examines bit 1 of the status byte; if it's set to 1, the program calls subroutine "Read_data".

Example: When an error occurs in the HP 8753 Network Analyzer, bit 3 of its status byte is set, and the error is placed in an error queue. The following program finds out if the error bit is set and, if it is, reads and prints the error. (The program is on your disk of examples as "8753_ERRS".)

```
10  !RE-STORE "8753_ERRS"
20  !
30  DIM Err$(50)
40  REPEAT
50  Stat = SPOLL (716)
60  UNTIL BIT (Stat,3) = 1
70  OUTPUT 716; "OUTPERRO;"
80  PRINT "HANDLING ERRORS"
90  ENTER 716;Err,Err$
100 PRINT Err, Err$
110 LOCAL 716
120 BEEP 600,.01
130 END
```

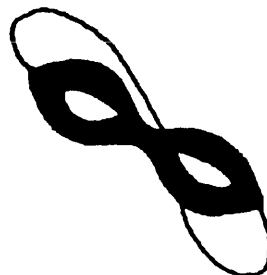
How it works: Lines 40-60 are a loop that repeats until bit 3 of the instrument's status bit is set. Line 50 reads the instrument's status byte into the variable Stat, while in line 60, if the error queue summary bit is not set, the program loops until it is set.

If the error queue has something in it, line 70 places the 8753 in remote mode and instructs it to send the error number and error message. Line 90 "reads" the error number into the variable Err, and the error message into Err\$.

Line 110 returns the HP 8753 to local mode so the front panel is available to the operator. Line 120 gives an audible signal that something is wrong.

To Mask or Unmask?

When you enable a bit in the interrupt register or read a bit of a status byte or another status register, what you are actually doing is creating a "mask" for the register. A mask opens a window on the bits you want to see, while suppressing undesired information.



Why is it called masking? Because it's actually the Boolean AND operation:

Register	Mask	Result (Register AND Mask)
0	0	0
0	1	0
1	0	0
1	1	1

You can see that when the mask is 1, the result of the register bit AND the mask bit can be 1. When the mask is 0, the result can never be 1 – that is, never enabled.

When you enable an interrupt, what you're actually doing is writing a mask. The mask is HP-IB status register 5, called the Interrupt Enable Mask. It masks (or un.masks, if you prefer) status register 4.

Thus, to unmask bits 1, 2, 3 and 4 of the interrupt status register, you need a mask that looks like this:

```
00000000
00011110
```

You do this, of course, with an HP BASIC statement:

```
ENABLE INTR 7; 30
```

What this statement actually does is to store the bit mask in HP-IB's Interrupt-Enable Register – which you know is register 5.

Most of the time you won't have to worry about masking, because it goes on behind the scenes. But it's a commonly-used computer term, and you should be aware of what's happening.

If you get confused, remember it this way:
A mask unmasks what you want to see.

Got it?

No Masking Necessary

Notice that you don't have to mask the status byte for a serial poll like you did for an interrupt. That's because you're actively polling the instrument – you're not waiting for it to generate an interrupt.

You can use the HP BASIC statement SPOLL to find out information about any instrument. Some instruments also have an instrument command string (such as OUTPSTAT for "output status") that allows you to get status information by using the OUTPUT statement. For example:

```
50 OUTPUT 716; "OUTPSTAT"
```

Review Quiz

1. Name three ways a computer can find out if an HP-IB instrument needs service.
2. You can enter and run the following mini-program:

```
10 STATUS 7,4; Stat  
20 PRINT Stat  
30 END
```

The result that's printed on the screen is:

```
66
```

Assuming interface code 7 is HP-IB, what bits are set in this status register. Is the "SRQ received" bit set?

3. In the section of code below, lines 100 and 110 cause an interrupt when a handshake or bus error occurs during an OUTPUT statement. If there are three such bus errors while the FOR-NEXT loop is being executed, how many times will the value for V be printed?

```
100 ON INTR 7 GOSUB Read_volts
110     ENABLE INTR 7; 64
120     FOR I = 1 TO 100
130     NEXT I
140     STOP
150 Read_volts:!  
160     ENTER 722;V
170     PRINT V
180     RETURN
190     END
```

4. Bit 6 of the HP 8753's event status register is "User Request." It is set when the operator presses a front panel key or turns the knob.
- Write a brief section of BASIC code that will enable bit 6 for reporting. If the user presses a front panel button, branch to a subroutine called "Button" that prints "Interrupted by LOCAL yokel!"
5. Change line 50 of the program "8753_ERRS" to two statements that use the HP BASIC OUTPUT and ENTER statements and one of the HP 8753's instrument command strings to "read" the status byte.

Saving and Reusing Instrument Data

Data manipulation, storage, and retrieval in HP BASIC *can* be a highly complex subject for study. There are many things you can do, many pointers to keep track of.

Added to which is the mysterious instrument tethered at the end of a cable, pumping in who-knows-what over HP-IB. It's enough to make you throw up your hands!

But if you just remember two little words – "unified I/O" – data storage and retrieval can be fairly simple. In part 3 of this course you'll learn some fancy formatting that'll speed up your programs and make them more efficient. In this lesson, though, you'll learn ordinary blue-collar methods to save data on discs, and re-use it later.

This lesson covers:

- Instrument data.
- Free-field format.
- Data OUTPUT to instruments.
- Data ENTERed from instruments.
- The number builder.
- Storing instrument data on disk.
- Retrieving data from disk.
- OUTPUT USING.

What Is Instrument Data?

As you know, there's a big difference between HP BASIC statements that make up your programs and *data*. Within the computer, there are three kinds of data:

- Real numbers.
- Integer numbers.
- ASCII strings.

You can usually think of instrument data as a long ASCII string or series of strings – unless you've specifically changed the format.

Data Formats

As you'll learn later in lessons 25 and 26, you can change the format of data.

The general formats are:

- 1. Free-field:** No format, or "K" specifier; standard number format. (This is the most common type.)
- 2. Free-field with attributes:** Attributes (FORMAT OFF, PARITY ON, etc.) specified in I/O path ASSIGN statement. Data transfer must conform to attributes.
- 3. Fixed-field:** Uses image (such as OUTPUT USING or PRINT USING). This overrides an I/O path ASSIGN statement.

In this lesson, you'll deal mostly with free-field format. But you should know that you can change the data format in the computer, by using images (ENTER USING, OUTPUT USING) or by specifying attributes when you ASSIGN path names.

In addition, some instruments, including the HP 8590A Portable Spectrum Analyzer featured in this lesson, can *deliver* data in different formats. In this lesson, though, you'll work with data that's in the most common format – ASCII characters.

Data Files

Don't confuse the data type with the three types of *files*:

- BDAT files.
- HP-UX files.
- ASCII files.

You can, for instance, store real numbers in an ASCII file. Or ASCII strings in BDAT files.

For free-field OUTPUT and ENTER operations, you can think of an instrument as a kind of ASCII *file*. The *data* are strings – of characters, real numbers, or integer numbers. INPUT and OUTPUT to and from an instrument are just like to and from an ASCII file.

(In part 3, lessons 25 and 26, you'll learn to manipulate data using images and ASSIGNED attributes that specify exactly what type it is.)

The default data representation used with devices like instruments is ASCII. So for now, assume all data is made up of ASCII characters.

Data OUTPUT

Getting commands to an instrument is usually pretty straightforward; you insert the command string in quotation marks in an OUTPUT statement, like this:

```
40 OUTPUT @Analyzer; "SNGLS;TS;"
```

Remember, *the exact format of the command string depends on the instrument*. In some cases you'll be able to separate commands with a space, other times (as in the 8590A) you'll need a comma or semicolon.

The data, of course, is in the form of a string of characters.

ENTERed Data

Just as with OUTPUT, all ENTER statements from instruments interpret the data as ASCII characters.

But, if you remember, you can't calculate with ASCII data; to the computer it's just a big mess of characters with no meaning. You can print it just as it comes in, of course. But to *do* anything with the data—calculate with it, add or subtract it, make a meaningful plot—you have to first turn it into either *integers* or *real numbers*.

You can do this by *either*:

- Using VAL\$ to calculate the value of each number after you've brought it in.
- Using ENTER to generate numbers as the data comes in.

"But wait!" you're thinking. "I've already used ENTER over and over. And I rarely so much as smelled a string variable."

It's true; most of your ENTER statements look like this:

```
100 ENTER 718; Frequency
```

or

```
250 ENTER @Voltmeter; Readings (*)
```

The reason you haven't had to worry about what kind of data is coming in is a wonderful HP BASIC feature known as the *number builder*.

The Number Builder

The number builder used with ENTER is a built-in firmware routine that evaluates incoming ASCII characters and then "builds" them into an integer or real number.

Your disk of examples has a file called DATA_17, that you can use to demonstrate the number builder. First, find out what's in the file:

1. Put the disk in the disk drive and address it using MSI, like this:

```
MSI ":,700,0"
```

(Within the quotation marks, substitute the address of the mass storage unit you'll use on your computer.)

2. Type and run this mini-program:

```
SCRATCH_↓  
EDIT_↓
```

```
10 DIM String$ [50]  
20 ASSIGN @Path TO "DATA_17"  
30 ENTER @Path; String$  
40 PRINT String$  
50 END
```

```
RUN_↓
```

The display shows what's actually in the ASCII string:

```
The number is 123.45e5abcde
```

The string is "The number is 123.45e5abcde". That's the full string; it could be data just as it's sent from an instrument. Since you specified a *string* variable (String\$), you see all the ASCII characters. It doesn't go through the number builder.

Now put the same data through the number builder: change lines 30 and 40 to specify a *numeric* variable:

EDIT 30,␣

```
30 ENTER @Path; Var  
40 PRINT Var
```

RUN,␣

This time you see a *number* :

```
1.2345E+7
```

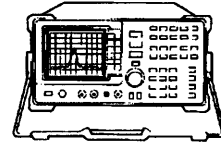
See what happened? When you put data into a numeric variable, it goes through the number builder. The number builder strips leading characters and non-numeric characters, leaving just the number in your numeric variable.

The number builder uses some fairly arcane rules to decide what it keeps and throws away. It sees numbers as being made up of the characters 0 through 9, ".", "+", "-", "E" and "e". Characters other than digits have to occur in meaningful positions.

There are many more number-building rules. But in general, you can let the number builder work for you without worrying about what it does. If the instrument is sending numeric data, then numeric data is what you'll get.

This Lesson's Featured Instrument: The HP 8590A Portable RF Spectrum Analyzer

The HP 8590A weighs just 30 pounds and is small enough to fit under an airplane seat, yet boasts many of the features of much larger models. It has a frequency range of 10 kHz to 1.5 GHz (1.8 GHz optional) and an amplitude range of -115 dBm to +30 dBm.



Operation is easy; you center the signal with a [FREQUENCY] control, resolve it with [SPAN], and move it up and down on the screen with [AMPLITUDE]. Functions such as [PEAK SEARCH] let you put a marker on displayed signal, then read the frequency and amplitude.

HP-IB Operation

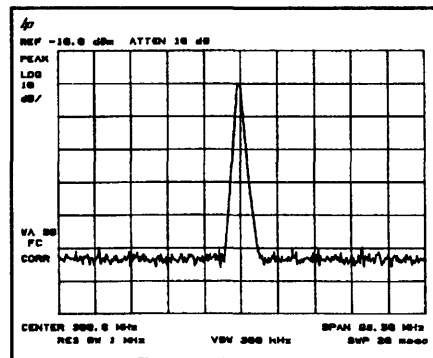
The HP 8590A's HP-IB codes are: SH1, AH1, T6, L4, SR1, RL1, PP0, DC1, DT1, C1, C2, C3, C28.

The analyzer has over 80 instrument commands you can use for controlling it. Here are a few of them:

CF

Center Frequency

Description: The CF command specifies the value of the center frequency.



IP

Instrument Preset

Description: The instrument preset command, **IP**, executes the following commands (this is an abbreviated list):

AUNITS DBM: Selects dBm amplitude units.

FA: Sets the start frequency. (0 MHz)

FB: Sets the stop frequency. (1500 MHz)

MDS W: Selects data size of one word, which is two 8-bit bytes.

MDS

Measurement Data Size

Description: The **MDS** command formats binary measurements:

B selects a data size of one 8-bit byte.

W selects a data size of one word, which is two 8-bit bytes.

MKCF

Marker to Center Frequency

Description: The **MKCF** command sets the center frequency equal to the marker frequency and moves the marker to the center of the screen.

MKPK

Marker Peak

Description: The **MKPK** command positions the active marker on signal peaks. Executing **MKPK HI**, or simply **MKPK**, positions the active marker at the highest signal detected. If an active marker is on the screen, the parameters move the marker accordingly.

SNGLS

Single Sweep

Description: The SNGLS command sets the analyzer to single-sweep mode. Each time single sweep is pressed or TS (take sweep) is entered, one sweep is initiated.

SP

Span

Description: The SP command changes the total displayed frequency range symmetrically about the center frequency. The frequency span readout refers to the displayed frequency range. Divide the readout by ten to determine the frequency span per division.

TDF

Trace Data Format

Description: The trace data format, TDF, command formats trace information for return to the controller.

M, measurement units, returns values in display units from -32768 to +32767.

P, parameter units, returns absolute measurement values such as dBm or Hz.

A returns data as an A-block data field. The MDS command determines whether data comprises one or two 8-bit bytes.

I returns data as an I-block data field. The MDS command determines whether data comprises one or two 8-bit bytes.

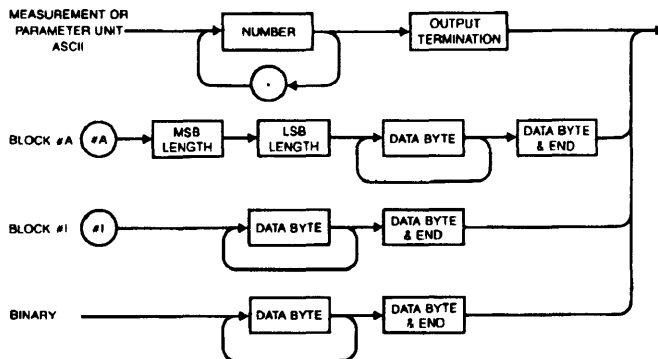
Specifying **B** enables binary format. The MDS command determines whether data comprises one or two 8-bit bytes.

A, **B**, **I**, and **M** are defined in the analyzer's internal amplitude units (log: hundredths of dBm; linear: 8,000 = top of screen and 0 = bottom of screen). **P** is in the current parameter unit specified by AUNITS.

TRA/TB

Trace Data Input/Output

Description: This command provides a method for reading or storing values into a trace. Adding a question mark (TRA?) queries the analyzer for trace data. The form of the query response is dependent upon the previously used Trace Data Format command as follows:



Storing Instrument Data

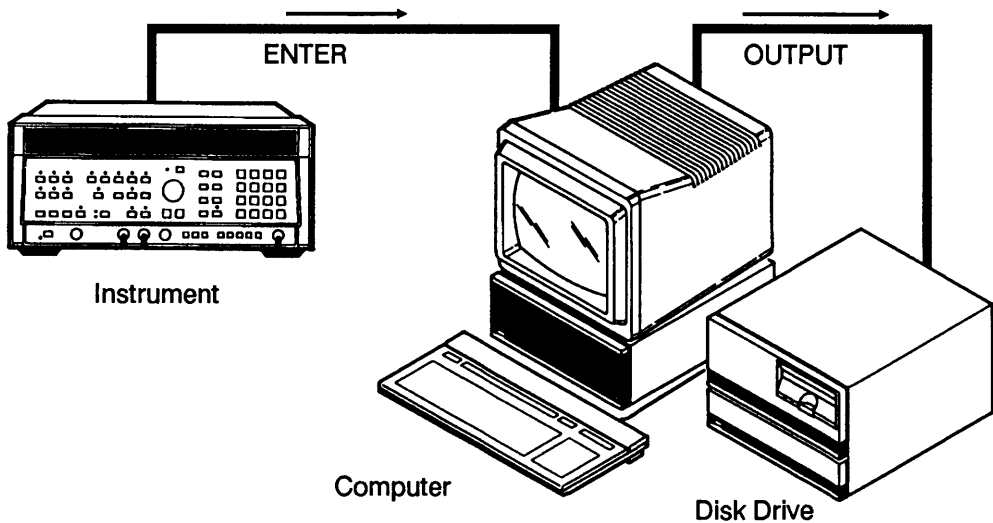
If you remember the techniques used to put ordinary data on mass storage such as disk or tape, you'll find storing data from instruments is the same. (And if you don't remember, sneak a peek at lesson 10 before continuing.)

For a file, you OUTPUT data to the file, then ENTER it from the file. With an instrument, though, you're more likely to ENTER data from the instrument, then OUTPUT it. The same cardinal rule applies, though: Read it the way you wrote it. Or, in the case of an instrument:

Cardinal Rule 

OUTPUT data the way you originally ENTERed it.

Saving data is just like storing any other kind, *except* you have to get the data from the instrument first.



Much simplified, the sequence of events, then, is:

1. Get data from the instrument.
2. Create a data file on the disk or other mass storage unit.
3. Output data to the file.

For manipulation, you'll probably want to store numeric data from instruments in BDAT files. That's because:

- ASCII files can't be accessed randomly; BDAT files can.
- ASCII data can't be formatted; binary data can.

Example: The program "8590_STORE" on your disk of examples shows how a 401-point trace from the HP 8590A Portable RF Spectrum Analyzer is stored on disk.

Here's the complete program:

```
10    !RE-STORE "8590_STORE"  
20    GOSUB Init  
30    GOSUB Get_data  
40    GOSUB Print_data  
50    GOSUB Create_file  
60    GOSUB Store_data  
70    STOP  
80    !  
90 Init: !  
100   REAL Trace_a(1:401)  
110   ABORT 7  
120   ASSIGN @Analyzer TO 718  
130   CLEAR @Analyzer  
140   RETURN  
150   !
```

```

160 Get_data: !
170   OUTPUT @Analyzer;"IP;"
180   OUTPUT @Analyzer;"TDF P;"
190   OUTPUT @Analyzer;"SNGLS;"
200   OUTPUT @Analyzer;"CF 300MZ;"
210   OUTPUT @Analyzer;"SP 200MZ;"
220   OUTPUT @Analyzer;"TS;"
230   OUTPUT @Analyzer;"MKPK HI;"
240   OUTPUT @Analyzer;"MKCF;"
250   OUTPUT @Analyzer;"TS;"
260   OUTPUT @Analyzer;"TRA?;"
270   ENTER @Analyzer;Trace_a(*)
280   ASSIGN @Analyzer TO *
290   RETURN
300   !
310 Print_data: !
320   PRINT Trace_a(*)
330   RETURN
340   !
350 Create_file: !
360 MASS STORAGE IS ":,700,0" ! (or ":,1500,0" for a language processor)
370   CREATE BDAT "TRACE_A",13
380   RETURN
390   !
400 Store_data: !
410   ASSIGN @File TO "TRACE_A"
420   OUTPUT @File;Trace_a(*)
430   ASSIGN @File TO *
440   RETURN
450   END

```

The program is an example of structured programming: every major step is a subroutine called from the main program. This makes changing or debugging the program quite easy.

The main program calls all the subroutines:

```
10 !RE-STORE "8590_STORE"  
20 GOSUB Init  
30 GOSUB Get_data  
40 GOSUB Print_data  
50 GOSUB Create_file  
60 GOSUB Store_data  
70 STOP
```

The Init routine sets up a 401-point real-number array, clears HP-IB, and assigns a path to the HP 8590A Portable Spectrum Analyzer.

```
90 Init: !  
100 REAL Trace_a(1:401)  
110 ABORT 7  
120 ASSIGN @Analyzer TO 718  
130 CLEAR @Analyzer  
140 RETURN
```


The Get_data subroutine sets up the analyzer.

```
160 Get_data: !
170 OUTPUT @Analyzer;"IP;"
180 OUTPUT @Analyzer;"TDF P;"
190 OUTPUT @Analyzer;"SNGLS;"
200 OUTPUT @Analyzer;"CF 300MZ;"
210 OUTPUT @Analyzer;"SP 200MZ;"
220 OUTPUT @Analyzer;"TS;"
230 OUTPUT @Analyzer;"MKPK HI;"
240 OUTPUT @Analyzer;"MKCF;"
250 OUTPUT @Analyzer;"TS;"
260 OUTPUT @Analyzer;"TRA?;"
270 ENTER @Analyzer;Trace_a(*)
280 ASSIGN @Analyzer TO *
290 RETURN
```

Line 180 sets the trace data format to produce data in absolute measurement values, such as dBm.

Lines 190-240 set the analyzer for single-sweep mode, 300 MHz center frequency, 200 MHz span; and puts one marker at the highest signal and another at the center frequency.

Line 260 queries the analyzer for its 401-point sweep. Line 270 then enters the data into an array, Trace_a.

The Print_data subroutine isn't necessary, of course, but it lets you verify the data as it comes in.

```
310 Print_data: !
320 PRINT Trace_a(*)
330 RETURN
```

Lines 350-370 specify the mass storage unit and create a binary data file on it.

```
350 Create_file: !
360 MASS STORAGE IS ":",700,0"
370 CREATE BDAT "TRACE_A",13
380 RETURN
```

To determine the number of records, multiply the 401-point trace by 8 bytes per point, then divide by 256 bytes per record. The result is rounded to the next largest integer, so you'll need 13 records for the file.

Finally, the data is stored on disk:

```
400 Store_data: !
410 ASSIGN @File TO "TRACE_A"
420 OUTPUT @File;Trace_a(*)
430 ASSIGN @File TO *
440 RETURN
```

Remember, assigning the path to an asterisk as in line 420 closes the path.

Retrieving Data

Once you've stored data on disk (or in other mass storage) you can turn off the computer, remove instrument cables, go on vacation—your data is preserved.

To get it back again, you:

1. Open a path to the file.
2. ENTER data from the file back into an array.

Here's another program (also on your disk of examples, as "8590_GET") that brings data back into the computer from mass storage:

```
10 !RE-STORE "8590_GET"  
20 REAL Trace_a(1:401)  
30 MASS STORAGE IS ":",700,0"  
40 ASSIGN @File TO "TRACE_A"  
50 ENTER @File;Trace_a(*)  
60 ASSIGN @File TO *  
70 END
```

Line 20 dimensions an array of real numbers, called Trace_a. Line 40 opens an I/O path to the file "TRACE_A" on the disk. Then line 50 brings data from the file into the array Trace_a. Once data is in the array in the computer, you can manipulate or plot it.

Using OUTPUT USING

You remember PRINT USING, don't you? The statement that let you format printed output the way you wanted it? You can do the same thing—use images—with OUTPUT and ENTER.

Why use images? You've seen how OUTPUT can control an instrument by sending it instrument commands. You have the instrument's manual in front of you, so you always know exactly what to send. Who needs OUTPUT USING or IMAGE, anyway?

Well, suppose your program prompts a user to input values for, say, frequency. Now you have a problem—because even if your prompts are very specific, you don't really know what the operator will type.

A frequency of 20 MHz may come in as any of these:

20
20.0
20 MHz
20.000000

This is a job for OUTPUT USING and IMAGE. With these HP BASIC statements, you can guarantee that no matter how the operator inputs a value, the instrument receives its command strings in the proper format.

Example: In his attempts to introduce automated testing at the Snagsby Corporation, industrial engineer Allan Woodcourt is plagued by the haphazard key entries of Simon Snagsby, a data entry operator and son of the owner.

Young Snagsby, you see, has never been able to entirely fathom units such as "kilo", "mega", and "giga". So when prompted to enter a frequency of 750 MHz, he is likely to play it safe and type "75000000 [ENTER]". How can Woodcourt make sure that Snagsby types the frequency in the correct MHz format?

Solution: Woodcourt uses an image statement that allows entry only in megahertz. Something like this:

```
10 ASSIGN @Analyzer TO 718
20 INPUT "Type the fundamental frequency in MHz", Fund
30 IMAGE "CF;", DDD.DD, "MZ;"
40 OUTPUT @Analyzer USING 30; Fund
```

The IMAGE statement in line 30, coupled with line 40's OUTPUT statement, force Snagsby to type a number with no more than three digits to the left of the decimal point (DDD). If he types a number that's too large, an error is generated, and the OUTPUT isn't executed.

In part 3, lesson 25, you'll learn about more things you can do by using IMAGEs with OUTPUT and ENTER.

Review Quiz

1. Data from instruments is usually in what form?
2. How can you convert ASCII data into numbers? (*Two ways!*)
3. Name two ways of changing the data format with HP BASIC.
4. A 401-point trace from the HP 8590A Portable RF Spectrum Analyzer is on your examples disk as 8590_TRACE. Write a program that gets the data from the disk and displays it on the screen.

Laboratory Exercise

Modify the program "8590_STORE" so that:

1. Data is read from the instrument into an array called `Data_array`.
2. Data is stored in a file called `8590_DATA`.
3. Data is stored in the file `8590_DATA` using an I/O path called `@Path`.

You'll find a solution on your disk of examples, as `SOL_LAB17`. You can list it and see how it works.

Making the Keyboard Work for You

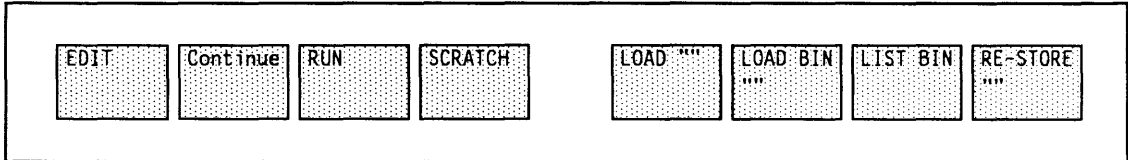
Up to this point, your use of the computer keyboard has been limited to the dedicated function keys and preprogrammed "soft" keys.

Now you're going to become a master of the keyboard instead of its slave. In this lesson you'll learn:

- The two ways to use softkeys.
- How to use LIST KEY to list the softkey definitions.
- How to change a softkey with EDIT KEY.
- How to use SET KEY to make a new key definition from a program.
- Storage of softkey definitions with STORE KEY.
- How to erase key definitions with SCRATCH KEY.
- A way to load keys with LOAD KEY.
- How to use ON KEY to generate a program interrupt.

How to Use Softkeys

When you first load HP BASIC into your computer, you see a list of labels across the bottom of the display. Here's an example:



Those are labels for the softkeys, keys [f1], [f2], [f3] (or [k1], [k2], [k3]), etc., on your keyboard.

In fact, depending on your combination of software and hardware, you may have as many as 24 softkeys. For instance, a Vectra or other personal computer using the HP BASIC Language Processor has three sets of user functions and each set has eight "keys". (You "cycle through" the system keys and the three different user menus with a combination of keyboard function keys and the SHIFT key.)

Note



For a description of the softkey functions at "wake-up", refer to your computer user manual.

Are these softkeys inviolate? Certainly not! In fact, you can give them your own definitions and labels—make them do *your* bidding. (On some computers, including a PC with the BASIC Language Processor, there's also a set of "system" softkeys that you can't change.)

With good softkey labels and definitions, you can make your programs much more elegant—and a lot easier to use, too!

Softkeys are used for either (or both) of two purposes:

- **Typing aids:** Pressing the key automatically "types" a series of keystrokes, just as if you'd typed it yourself.
- **Program branches:** Pressing the key interrupts a running program, causing it to branch or halt execution.

Softkeys as Typing Aids

When you look at the label for a softkey, what you actually see is *part* of the actual keystrokes that key automatically "types" when you press it. To see more information, you can list the softkeys.

Listing the Softkeys

To list the softkeys, use the LIST KEY statement. Unless you specify otherwise, the listing is to the current PRINTER IS device.

For instance, to list the keystrokes for all softkeys on the CRT screen, type:

```
PRINTER IS 1_␣  
LIST KEY_␣
```

Here's an example of a listing for softkey 1. (It may be different on your computer):

```
Key 1:  
System key: #  
EDIT
```

The listing shows the softkey number, and any keys "typed" by the softkey. (The # character here is actually the system key [CLEAR LINE]). In these listings, system keys such as [ENTER], [CLEAR LINE], and [INSERT CHAR] are shown as "System key:" followed by a special character. You'll learn more about these in a moment.

You can see that when you press softkey 1, it first "types" [CLEAR LINE], then it types the characters E D I T.

You'll probably want to list to a printer because the list of keys can be three or four pages long.

For example:

```
LIST KEY #26_↵
```

This lists all softkey definitions on a printer at address 26.

Or:

```
LIST KEY # 701_↵
```

This lists the current key definitions on a device (probably a printer) at address 701.

Rewriting a Softkey Definition

To change one of the softkey definitions, you can use the EDIT KEY statement, followed by the number of the softkey, like this:

```
EDIT KEY 2_↵
```

This displays the definition for softkey [f2] and lets you write a different definition if you want.

Example: The clockwatching Richard Carstone wants a softkey redefined so it gives him the time with a single keystroke.

Carstone knows about the `TIME` function that returns the current time when he types:

```
TIME$ (TIMEDATE) ↵
```

```
11:14:02
```

That's *way* too many keystrokes for the slow-fingered Carstone to type, though. To help Carstone, and make this function accessible from softkey [f2], type:

```
EDIT KEY 2 ↵
```

(Or just press the [EDIT] key followed by the [f2] softkey and [ENTER].)

The screen displays the current [f2] definition on the keyboard input line.

It also shows you a system message that indicates you can modify the softkey definition – something like this:

```
Ⓚ # STEP
```

```
Editing key 2
```

Now press and type the keys shown here:

```
[CLEAR LINE]
```

```
TIME$ (TIMEDATE)
```

```
[CTRL] [ENTER] (Press these two keys together.)
```

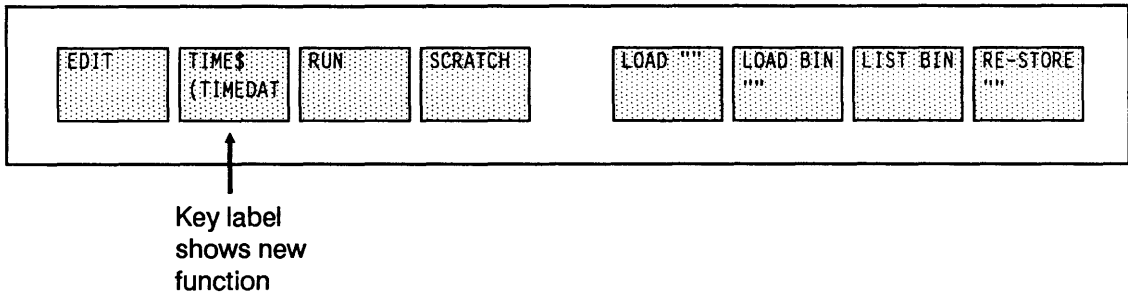
What you're actually doing is writing the softkey's new definition – what it will "type" when you use it. Pressing the [CTRL] key at the same time as the system key [ENTER] (or [RETURN], or [END LINE], etc.) tells the computer not to *execute* that key now, but to make it part of the softkey program. So it substitutes an "E" for the [CTRL] [ENTER]. You see the entire "program" now:

```
TIME$ (TIMEDATE)␣E
```

Now press the "real" [ENTER] key to end the line:

[ENTER]

You can see that the label for softkey [f2] changes immediately to show its new function:



Press softkey [f2] to see the time:

```
11:14:05
```

A softkey definition is essentially one typed line. So you should make sure never to exceed the maximum length of your display's keyboard input line – usually 160 characters.

What happened to the softkey label? The label you see is actually just a "window" onto the first few keystrokes of the key's definition:



← You see the first part of the soft key definition in its label

In this case (depending on your computer) you see most or all of the TIME\$ (TIMEDATE) keystrokes.

But if your definition is long, you won't see it all. Only the first few keystrokes are displayed.

One other thing: if you begin a sequence of keystrokes with a system key, such as [CLEAR LINE], you won't see that key displayed in the label.

In its infinite wisdom, the computer realizes that you probably don't want leading system keys such as [CLEAR LINE] to appear in the label. So it simply doesn't display them.

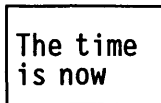
For instance, take a series of keystrokes such as this:

~~[CTRL] [CLEAR LINE]~~ The time is now
 [CTRL] [CLEAR LINE] TIME\$ (TIMEDATE)
 [CTRL] [ENTER]

These keystrokes might show up as:



← Leading system key not displayed



← You see only these characters in the label



TIME\$(TIMEDATE)

← Not displayed



The added spaces after the words "The time is now" give you a "clean" label. They ensure you don't see subsequent characters of the definition.

Changing a Key from a Program

EDIT KEY works well for "on-the-spot" softkey definitions. But when you want your program to automatically change the softkey definition, use a SET KEY statement.

To use SET KEY, you assign the string of characters to be "typed" by the softkey to a string variable. Then you use SET KEY to assign the variable to a softkey. Look at this simple illustration:

```
10 A$ = "SIN"  
20 SET KEY 5, A$  
30 END
```

Line 10 assigns the characters SIN to the string variable A\$. Then line 20 assigns that string to softkey [f5].

After running this section of code, you'd see the characters SIN in the softkey 5 label. And pressing [f5] would "type" the characters:

```
SIN
```

Example: Tired of remembering—and typing— long strings such as MSI ":",1500,0" every time you want to change the default mass storage unit? The example program "MSI_KEYS" lets you use softkeys to change MSI.

Load and run the program MSI_KEYS from the disk of examples (or type it in following the code below).

```
LOAD "MSI_KEYS" ↵  
LIST ↵
```

```

10 !RE-STORE "MSI_KEYS"
20 DIM A$ [50]
30 DIM B$ [50]
40 A$=CHR$(255) &"#&"Floppy disk "&CHR$(255) &"#&"MSI":,1500,0""&CHR$(255) &"E"
50 B$=CHR$(255) &"#&"Hard disk "&CHR$(255) &"#&"MSI":,1500,2""&CHR$(255) &"E"
60 SET KEY 4,A$
70 SET KEY 5,B$
80 END

```

Note



This program was written for a Vectra or other PC with a single floppy disk at ":",1500,0" and a hard disk at ":",1500,2". If your MSI addresses are different, you'll have to change lines 40 and 50 to the new addresses—say ":INTERNAL" and ":",700". Naturally, you'll want labels such as "Internal DD" and "External DD" instead of "Hard disk" and "Floppy disk", too.

How it works: Lines 20 and 30 create memory space for the strings A\$ and B\$. Line 40 uses "&" to join several characters and strings. It assigns the following string of keys to the variable A\$:

```

[CTRL] [CLEAR LINE] Floppy disk
[CTRL] [CLEAR LINE] MSI ":",1500,0"
[CTRL] [ENTER]

```

In the string, the expression CHR\$(255) is how you generate the [CTRL] key. The pound sign (#) after [CTRL] is how you generate the system key [CLEAR LINE].

The extra spaces after "Floppy disk" guarantee that only those characters appear in the label.

In order to have the key "type" the string MSI ":",1500,0", you need to enter *two* quotation marks everywhere the string is to contain one quotation mark. So it looks like this:

```
"MSI" " : , 1500 , 0 " " "
```

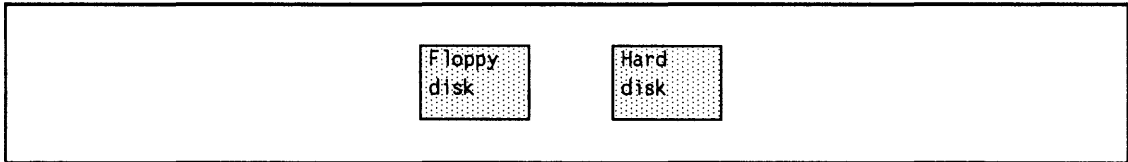
The expression CHR\$(255) &"E", of course, is our old friend [CTRL] [ENTER].

Line 50 is similar to line 40, except it's for the hard disk instead of the floppy.

Line 60 assigns string A\$ to key [f4], while line 70 assigns string B\$ to key [f5].

Try it! Type (or press):

```
RUN_
```



You can see that softkeys [f4] and [f5] have new labels:

You can now use [f4] and [f5] to switch the current MSI between two mass storage units.

Common System Keys

Here are some common system keys and the characters you'll see when you use them with the [CTRL] key, or with CHR\$(255). For a complete list, look in your HP BASIC reference (or condensed reference) material under *Second Byte of Non-ASCII Key Sequences*.

Character	System Key
#	[CLEAR LINE]
%	[CLR END]
*	[INSERT LINE]
+	[INSERT CHAR]
-	[DELETE CHAR]
/	[DELETE LINE]
<	[←]
>	[→]
B	[BACK SPACE]
E	[ENTER]
G	[SHIFT] [→]
H	[SHIFT] [←]
I	[CLR I/O]
K	[CLEAR DISPLAY]
L	[GRAPHICS]
M	[ALPHA]
U	[CAPS LOCK]

Storing Softkey Definitions

Just like using STORE and RE-STORE to store programs, you can use STORE KEY and RE-STORE KEY to save softkey definitions in a file on mass storage.

For example, to save the current softkey definitions as a file called MY_KEYS on the current mass storage unit, remove the disk of examples, insert an initialized disk, and type:

```
STORE KEY "MY_KEYS" _
```

The key definitions are stored as a BDAT file on the disk or other MSI.

Erasing Softkey Definitions

Use SCRATCH KEY to erase all current softkey definitions. Type:

SCRATCH KEY_↵

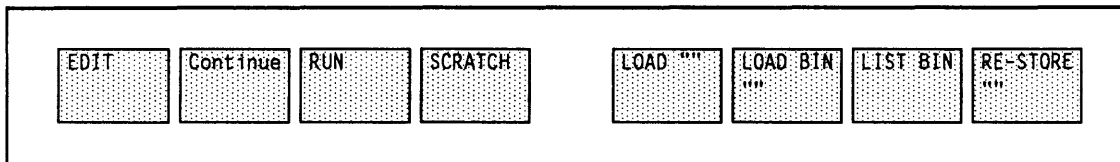
You can see all user softkeys and their labels are erased.

Not to worry, though! You have them in mass storage, remember?

Loading Softkeys

To get the *default* user softkeys again, just type LOAD KEY. Try it:

LOAD KEY_↵



You can see all the original softkeys are back again.

What about your own keys? To reload softkeys from a file, use LOAD KEY with the file name, like this:


LOAD KEY "MY_KEYS"_↵

Presto! Your softkeys are back in place, ready for you to use them again.

~~Editing Softkeys on a PC:~~

If you're using a Vectra or other personal computer equipped with the HP BASIC Language Processor, EDIT KEY allows you to enter most keys into a softkey definition.

Most, but not all.

You may have noticed some differences already. For instance, on some monitors the [CTRL] key doesn't appear as a reverse-video . And there's more:

Exactly what you can and cannot enter using EDIT KEY depends on your keyboard and the version of software for the Language Processor. In general, though, you won't be able to enter key combinations that require three keys on the personal computer.

For example, on a Vectra keyboard, you normally press [SHIFT] [DEL] to delete a line. As you remember with EDIT KEY, it seems as if you'd simultaneously press: [CTRL] [SHIFT] [DEL].

Since you can't enter three keys using EDIT KEY, though, this combination doesn't work.

Naturally, you can use SET KEY normally, to put *any* combination of keystrokes into a softkey—even on a Vectra or other PC.

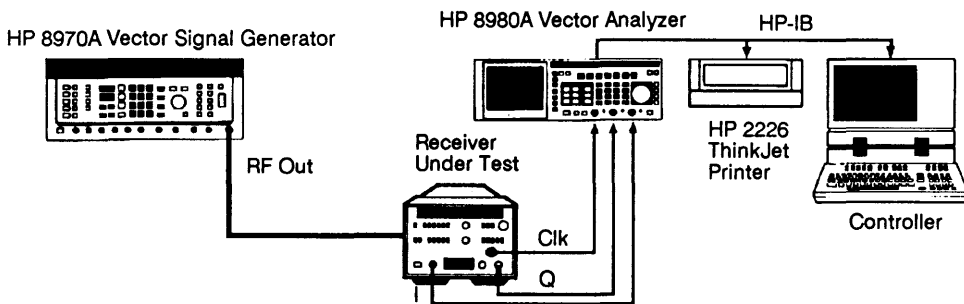
Or else you can use EDIT KEY; just keep in mind that you won't be able to enter some key combinations this way.

This Lesson's Featured Instrument: The HP 8980A Vector Analyzer

The HP 8980A is similar in concept to a two-channel sampling oscilloscope. Unlike general-purpose scopes, though, the Vector Analyzer has 350 MHz X-Y (vector) bandwidth, and excellent I/Q channel matching. It also has a 12-bit A/D converter and many specialized functions, the result of its own internal software.

The HP 8980A provides a real-time visual display of phase and magnitude, making it highly useful in evaluating radar receivers, or looking at the baseband I/Q signals of digital communications systems.

In a typical application, calibrating a radar or digital microwave receiver, a signal generator supplies a test signal. The HP 8980A Vector analyzer displays quadrature, gain, and dc errors. You adjust the receiver while watching the results of the vector display.



The instrument's Auto Scope function automatically scales voltages, offsets, timing, and trigger for the best view of a signal.

HP-IB Capabilities

Default HP 8980A address is 9. All functions are programmable except the power switch and a few other functions not needed for remote operation. The HP 8980A's interface capabilities are: SH1, AH1, T5, TE0, L3, LE0,SR1, RL1, PP1, DC1, DT0, C0, E2.

The HP 8980A has lamps for RMT, LSN, TLK, and SRQ. You can configure the 8980A for talk/listen or talk only. Talk/listen allows the instrument to both transmit and receive data over HP-IB. Talk only, of course, sets the instrument to transmit data only.

To switch back and forth, you press the [HP-IB] button on the 8980A's front panel. Then you press the applicable softkey on the 8980A front panel. (Yes, some instruments have user-definable keys, too.) In this lesson, leave the instrument in talk/listen.

Here are some selected instrument command strings, sent over HP-IB with the HP BASIC OUTPUT statement. Many strings can be used either normally, or as queries (with a question mark – for example "ERR?"). A query interrogates the Vector Analyzer as to its current state.

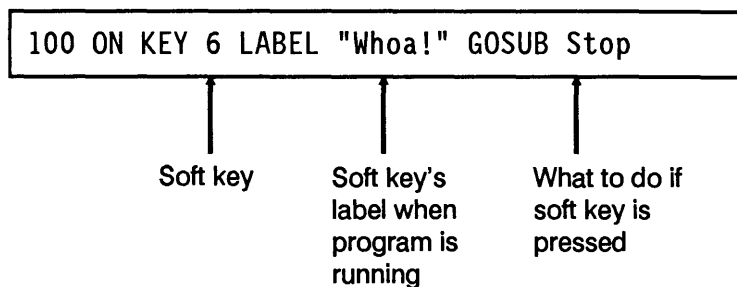
CMD	Description
*RST	Performs an HP 8980A reset, and sets the instrument to its PRESET condition. Also clears all pending operations.
AUT	Causes the HP 8980A to execute an Auto Scope function. Takes several seconds. Alternate mnemonics: AUT, AUTOSCALE, AUTOSCOPE. Example: AUT;

CMD	Description
DISP	This command selects the DISPlay subsystem. This subsystem controls markers, display mode, split screen, and signal intensity.
MODE	Used to set and query the display mode that is to be used. Query responses and preferred command data: CHANI, CHANIQ, CHANQ, CONALIGN, CONST, THREED, VECALIGN, VECTOR. Examples: DISP:MODE VECTOR; DISP:MODE? ASCII input and response.
ERR?	Used to query the next error in the error queue. The error queue is 16 registers deep, and operates on a first-in, first-out basis. A "0" is returned if there are no further errors. A large negative number returned indicates there has been an error queue overflow. Returns numeric integer data.

Keys for Program Branches

Lesson 16 (you *did* do lesson 16, didn't you?) explained what a program *interrupt* is. Remember, an interrupt branched execution if (and only if) it occurred.

You can use a softkey the same way—to branch execution when it's pressed. The secret is the ON KEY statement, and it looks like this:



The ON KEY statement lets you specify what softkey you'll use, what the key's label is, and what action to take—generally a GOSUB, GOTO, or CALL.

To try a simple example, type:

```
SCRATCH_␣  
EDIT_␣
```

```
10 I=0
20 ON KEY 5 LABEL "Whoa!" GOTO Halt
30 LOOP
40 I = I+1
50 PRINT I
60 END LOOP
70 Halt: PRINT "Saved by f5!"
80 END
```

Now run this mini-program:

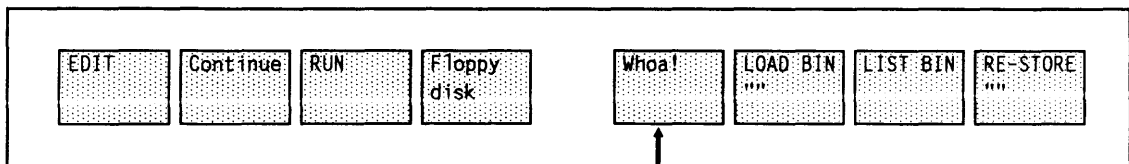
RUN_␣

The program takes off like a sprinter, filling the screen with the current value of I:

```
1
2
3
```

Like the sorcerer's apprentice of fairy-tale fame, you seem powerless to halt the program – except that you've built in an "escape" with your ON KEY statement.

Look at the softkey labels. They're just as you specified earlier – except for key [f5].



Label changed while
program is running

The label for [f5] is changed to "Whoa!". Press it to stop the program:

Saved by f5!

How it works: Line 20 "sets" softkey [f5]. After line 20 is executed, the computer remembers what it must do: "If softkey [f5] is pressed, I branch immediately to the label Halt."

If you don't press [f5], the computer goes through its loop endlessly (unless you stop it with [PAUSE] or [RESET] or some equally inelegant and heavy-handed solution). It *does* prompt you with the softkey label while it's running, though.

When you press [f5], execution branches immediately to the label "Halt" (line 70), and execution halts.

Notice that when the program is finished, the softkey reverts back to its duties as a typing aid. Remember:

- Softkeys can be used as *typing aids* whether or not a program is running.
- ON KEY is active only in a running program.

Example: This following program uses one softkey to set up the HP 8980A Vector Analyzer and scale it for a meaningful display, and another softkey to read errors that may have occurred.

```
10 ASSIGN @Hpib to 7
20 ASSIGN @Va TO 709
30 ABORT @Hpib
40 CLEAR @Va
50 LOOP
60 ON KEY 1 LABEL "ERRORS" GOSUB Ersub
70 ON KEY 2 LABEL "SETUP" GOSUB Setup
80 END LOOP
90 Setup:
100 CALL Clear_screen
110 OUTPUT @Va;"*RST"
120 OUTPUT @Va;"AUT"
130 OUTPUT @Va;"DISP:MODE CHANIQ"
140 RETURN
150 Ersub: !
160 CALL Clear_screen
170 OUTPUT @Va;"ERR?"
180 ENTER @Va;Error_num
190 PRINT Error_num
200 IF Error_num <>0 THEN 170
210 RETURN
220 END
230 SUB Clear_screen
240 OUTPUT 2 USING "#,B";255,75
250 SUBEND
```

In this case, the two softkeys ([f1] and [f2]) are specified *within* the loop.

Line 240 in the Clear_screen subprogram could be replaced by a CLEAR SCREEN statement. However this shows another way of clearing the screen; it works for older versions of HP BASIC (before version 5.0) that lacked the CLEAR SCREEN statement.

Using ON KBD

The ON KBD statement causes a branch when you press almost *any* key. Here's an example:

```
410 LOOP
420 ON KBD GOTO Halt
430 END LOOP
440 Halt:END
```

This loop keeps the program running until you press a key; then the program ends.

Most keys except major system keys cause a branch with ON KBD. For maximum effect, add the keyword ALL, like this:

```
420 ON KBD ALL GOTO Halt
```

Now, pressing any key except [RESET], [SHIFT], or [CTRL] causes a branch.

Review Quiz

1. What are the two ways softkeys can be used?
2. Not content with viewing only the time at the press of a key, the clockwatching Richard Carstone also wants to have the date similarly handy.

Write a mini-program that lets Carstone press softkey [f8] to see the date. Give the key a clean label, like this:

```
Today's
Date
```

Hint: The date is shown by the function
DATE\$(TIMEDATE).

3. Modify the softkey definition for LOAD " " shown here so that it copies a file from the current MSI to ":",1500,0".
For example:

COPY "(File name)" TO ",1500,0"

```
Key 5:  
System key: #  
LOAD "  
System key: H  
System key: >  
System key: >  
System key: >  
System key: >  
System key: >  
System key: >  
System key: >  
System key: >  
System key: %  
System key: <  
System key: +
```

Data on Display

There's a whole world awaiting you in this lesson—a world of graphics. Charts, illustrations, graphs, tables, all can be created using HP BASIC. And if you happen to have a color monitor, printer, or plotter, your data and displays can *really* dazzle!

Sadly, though, you'll get only a glimpse of the wonders of graphic displays. You see, just as HP BASIC contains dozens of powerful statements specifically for instrument control, it also features a phalanx of graphics statements. So many, in fact, that graphics deserves its *own* complete self-paced study course.

In this lesson, then, you'll learn the rudiments of graphics for instrument control, along with some specialized graphics routines. You can drop these "cookbook" sections of code into your programs; they'll display and print your data in meaningful ways.

You'll learn about:

- The graphics display.
- GRAPHICS ON and GRAPHICS OFF.
- ALPHA ON and ALPHA OFF.
- CLEAR and GINIT.
- GDU's and UDU's.
- DRAW and PLOT and MOVE.
- PEN DOWN and PEN UP.
- Choosing pen type with PEN.

- Choosing LINE TYPE.
- Specifying the plotter with PLOTTER IS.
- FRAME.
- Specifying the VIEWPORT.
- WINDOW and SHOW.
- CLIP ON and CLIP OFF.
- AXES and GRID control.
- Creating labels with LABEL, LORG, LDIR, and CSIZE.
- Plotting data.
- Dumping to a printer with DUMP DEVICE IS and DUMP GRAPHICS.

The Graphics Display

Every lesson in this course – up to now, that is – has used what’s called the *alpha* display: What you type or print is shown as alphanumeric characters. When you first load BASIC, it automatically "wakes up" showing the alpha screen.

There’s another display hidden away, though: the *graphics* screen. You use the graphics display to show things such as plots, charts, and drawing with labels.

Showing the Graphics Plane

In most computers, the alpha and graphics displays occupy different portions of memory, so you can think of them as being on two different planes. To show the graphics plane, use the GRAPHICS ON statement:

```
GRAPHICS ON_
```

When you type this, you may or may not see graphics on your screen – depending on what’s in your computer’s graphics memory right now.

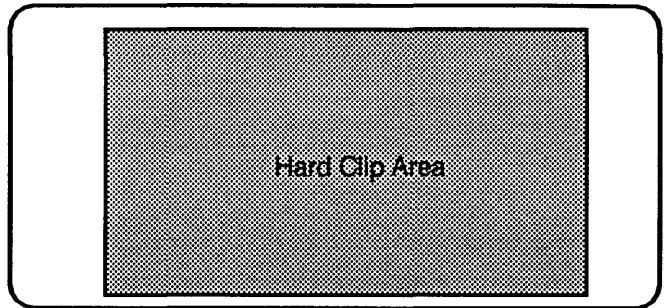
To see the area your screen has available for graphics, type:

GCLEAR_

GINIT_

FRAME_

You'll see a square drawn on your computer's CRT screen.



This is the *hard clip area*. You can't put graphics outside of it.

Note

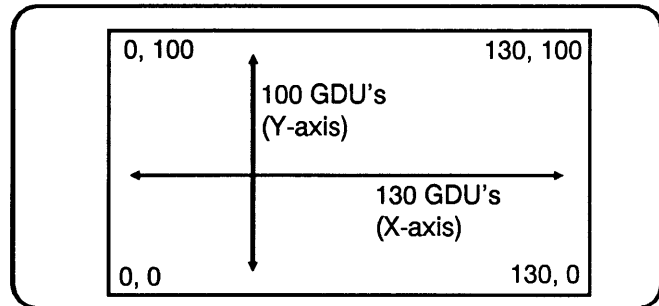


If graphics won't stay on your screen, press the [Graphics] or [Graph] key. On some computers with the BASIC Language Processor, you'll have to press the [Graphics] key to return to the graphics screen after most programs and *any* keyboard operation.

Locations in the Graphics Area

The area you see is divided into arbitrary measuring units, called GDU's (for *graphic display units*). Depending on your display, the graphics screen is usually about 130 GDU's wide (0-130) and 100 GDU's high.

You can think of it as 130 GDUs along the x-axis, and 100 along the y-axis.



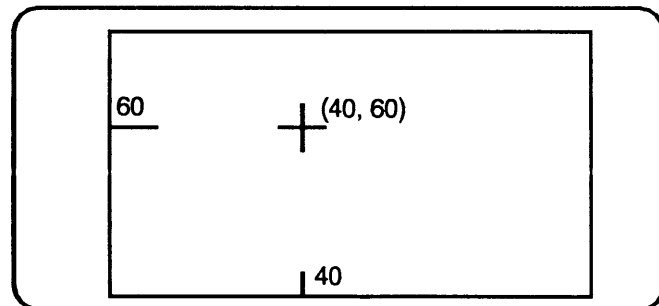
Another Note



On some displays, you may not be able to see the entire viewable area at once. Don't worry, though – you can set your viewport so it displays only on the screen. See the heading "Set the Viewport" a few pages after this.

To specify any location within the graphics area, you use its x,y coordinates (just as if it were a map). So position 0.0 is the lower left-hand corner. Position 40, 60 means:

- 40 GDUs along the x-axis, and
- 60 GDUs along the y-axis.



Incidentally, a GDU is an entirely arbitrary unit; it's not related to resolution, or the actual number of dots (pixels) on a CRT screen.

Graphics Fundamentals

Before you begin to plot data on the screen, take a few moments to learn about some fundamental graphics operations.

Initializing Graphics

Graphics initialization is a job for GINIT. This statement (it means "graphics initialize") sets most parts of the graphics plane back to initial values and locations. It's a good way of getting to *known* conditions before you begin.

Here's a *partial* list of what GINIT does:

```
PEN 1
CLIP OFF
CSIZE 5, 0.6
LDIR 0
LINE TYPE 1
LORG 1
MOVE 0,0
```

You'll learn more about what these mean later in this lesson.

Clearing Graphics

Notice that GINIT doesn't erase the graphics screen. Neither does GRAPHICS OFF. Another statement, GCLEAR, erases graphics for you.

To clear the graphics display, type:

```
GCLEAR_
```

This erases everything from the graphics plane, giving you a clean slate to draw on.

Drawing with the Pen

HP BASIC was created for use with, among other devices, HP plotters. That's why when you "draw" on the graphics area, you use statements that make it seem as though you're working with a plotter.

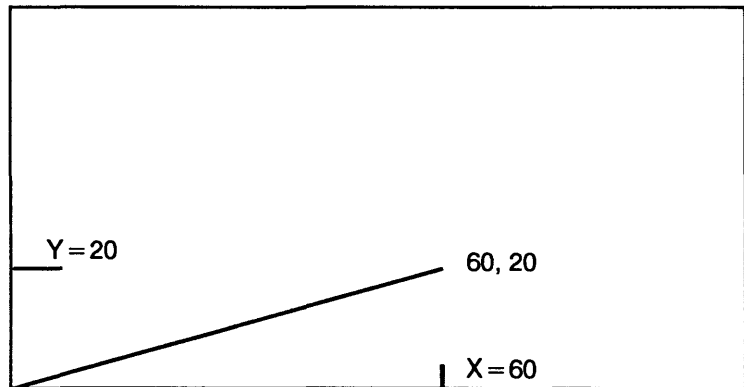
For instance, you "draw" with a "pen." To draw with the pen, you can use the DRAW statement. This draws a line from the current position to the one you specify.

To move the pen without drawing, use MOVE.

Try this example. Type:

```
MOVE 0,0↵  
DRAW 60,20↵
```

This *moves* the pen to location 0,0. Then it *draws* a line from there to location 60, 20.



Choosing Pen Type

After GINIT or at "wake-up," the pen type is 1. This *draws* a line.

You can also use other PEN statements, such as PEN 3 or PEN - 2. These are interpreted in different ways.

PEN > 0 draws
PEN = 0 complements
PEN < 0 erases

(On a color monitor or plotter, of course, the PEN numbers also specify color.)

To see how this works, enter the following mini-program:

SCRATCH_↵
EDIT_↵

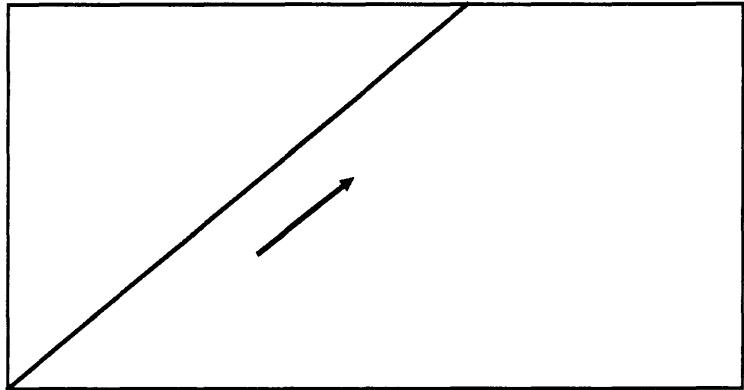
```
10 FOR I=1 TO 100  
20 DRAW I,I  
30 WAIT .1  
40 NEXT I  
50 END
```

Now press: [PAUSE] to get out of edit mode.

Then type:

GCLEAR_↵
GINIT_↵
FRAME_↵
RUN_↵

You can see a line is drawn from 0,0 to 100,100:



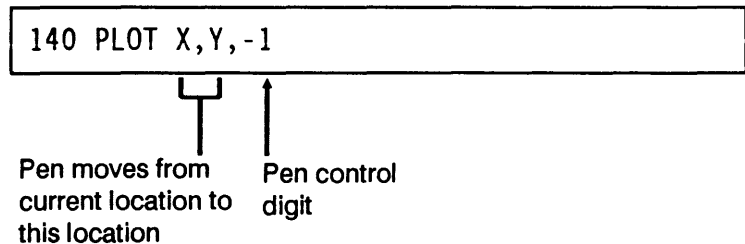
Now type:

```
MOVE 0,0_↵  
PEN-1_↵  
RUN_↵
```

You can see what happens. By changing the pen type to PEN-1, then redrawing the line from 0,0, you erased the original line.

Using PLOT

The PLOT statement is like DRAW—it lets you draw or plot a line from the current pen position to the specified one. PLOT is more sophisticated, though—it also lets you specify pen control and, under certain conditions, plot entire arrays. Here's how you use it:



The pen control digit works like this:

Pen Control Digit	Action
- Even (-2, -4, etc.)	Pen up before move
- Odd (-1, -3, etc.)	Pen down before move
+ Even (1,3, etc.)	Pen up after move
+ Odd (2,4, etc.)	Pen down after move

If you don't specify pen control, +1 is assumed (pen up after move).

Try an example. Change line 20 of your mini-program to a PLOT statement:

```
EDIT 20_↓
```

```
20 PLOT I, I
```

Then clear the screen. Press: [PAUSE]. Then type:

```
GINIT_↓
```

```
GCLEAR_↓
```

```
FRAME_↓
```

(GINIT, remember, moves the pen location back to 0,0, and selects PEN 1.)

Now run the program:

```
RUN_↓
```

And the result is...the same line is generated.

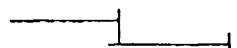

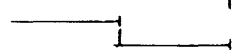

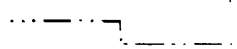

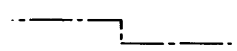
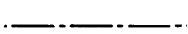
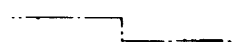
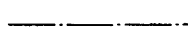
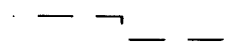
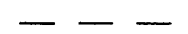
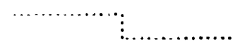
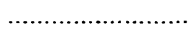
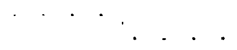



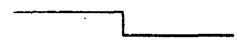

Choosing Line Type

To help differentiate the parts of your plot or graph, you can choose different *line types* for operations such as DRAW and FRAME.

Use the LINE TYPE statement, along with a number, to choose the type of line, like this:

```
LINE TYPE 3
```

Here are the different line types:

	LINE TYPE 10	
	LINE TYPE 9	
	LINE TYPE 8	
	LINE TYPE 7	
	LINE TYPE 6	
	LINE TYPE 5	
	LINE TYPE 4	
	LINE TYPE 3	
	LINE TYPE 2	
	LINE TYPE 1	

The last line type specified stays active in a program unless you change it.

Try this. Type:

```
GCLEAR_␣  
GINIT_␣  
LINE TYPE 5_␣  
RUN_␣
```

The line is plotted with the new line type.

Turning Graphics OFF

To turn the graphics display off, type:

```
GRAPHICS OFF_
```

Even though you "turn off" graphics with GRAPHICS OFF, whatever is drawn on the graphics plane remains – to see it again, just use GRAPHICS ON or the [Graphics] key.



HP 9000 Series 300 computers normally display the alpha and graphics planes together. You can't turn the graphics plane off at

GRAPHICS ON_

Showing the Alpha Plane

You use the ALPHA ON and ALPHA OFF statements (or the [Alpha] key) to turn the alphanumeric display on and off:

```
ALPHA ON_
```

```
ALPHA OFF_
```

On some (not all) computers, the alpha and graphics screens can *both* be on at once:

```
ALPHA ON_
```

```
GRAPHICS ON_
```

Oh, and one other thing: You *have* to turn graphics on and off with statements, but the computer defaults to alpha whenever anything is sent to the alpha display.

This means that pressing any key, or running a program, gives you the alpha display. So on computers that can't display both alpha and graphics together, it'll seem like you can't ever *examine* the graphics screen; it always defaults to alpha.

Don't worry, though. Everything from the alpha or graphics screen is stored when you specify ALPHA OFF or GRAPHICS OFF, so you don't lose anything.

As you work through this lesson, if you try graphics statements only to have the alpha display return (after a fleeting, tantalizing display of the graphics screen), press the [Graphics] key to "freeze" graphics.

A Typical Graphics Application

Perhaps the easiest way to learn about graphics for instrument control is to work through a typical application: taking data and plotting it on the screen, then printing or plotting it to set a hard copy.

Here's a checklist to follow to create useful graphics for most instrument control applications:

1. Know your instrument and data.
2. Initialize the graphics plane.
3. Set the viewport.
4. Scale the plotting area.
5. Put in axes lines.
6. Put in a grid.
7. Label your plot.
8. Plot your data.
9. Dump to a printer.

You'll learn how to do each of these – in order, of course.

Know Your Instrument

The first thing you need is at least a *general* idea of what the data from your instrument will be. Will there be negative and positive numbers? Will it be integers or real (full-precision) numbers? What will be the maximum and minimum values?

Since it's difficult to predict what *your* data will be, for purposes of this illustration you'll work with the famous "Turveydrop Trigolator." Invented by one Thomas Turveydrop as part of a failed experiment in perpetual motion, the Trigolator has a highly unique characteristic: no matter what amplitude, frequency, phase, or number of signals are applied to it, the Trigolator responds by producing the trigonometric functions sine and cosine of angles from 0 to 360 degrees.

This means that to plot a sine function, one set of data (the angles) is from 0 to 360; another set (the sines) is from 0 to 1 in size.

Initialize the Graphics Plane

When creating a new plot or drawing, the first thing you want to do is to initialize the graphics plane – clear graphics from the screen, set all graphics elements to standard, known locations, maybe put a frame around the graphics area, etc.

Specify the Plotter

Your initializing routine needs a PLOTTER IS statement to tell it where to make its plot. Type:

```
PLOTTER IS CRT, "INTERNAL" ̀
```

This uses the computer's internal CRT.

Turn Graphics On

The next thing you might want is to turn on graphics and turn off alpha. Type:

```
GRAPHICS ON_␣  
ALPHA OFF_␣
```

Clear and Initialize

Next you'll initialize graphics. Type:

```
GINIT_␣  
GCLEAR_␣
```

Use FRAME

It's not mandatory, but often you'll want to know just how big your usable graphics area is. You know how to do that, don't you? Of course! You just type:

```
FRAME_␣
```

FRAME puts a frame around the last area or sub-area you specified. (You'll see in a moment how to create sub-areas.)

Set the Viewport

The graphics area you see now is the *entire* area available for graphics. You can't draw anything beyond the limits of that frame. (It's sometimes called the hard clip area.)

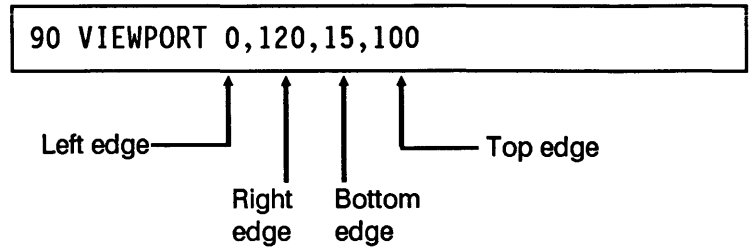
Often it's not convenient to use that entire area for a graph. You may want to leave space for the alpha or softkey display, or to put in some labels later.

To define the user graphics area, reach into your bag of tricks and pull out VIEWPORT.

Using VIEWPORT

The VIEWPORT statement lets you define a smaller plotting area — a "sub-area" within the full graphics area.

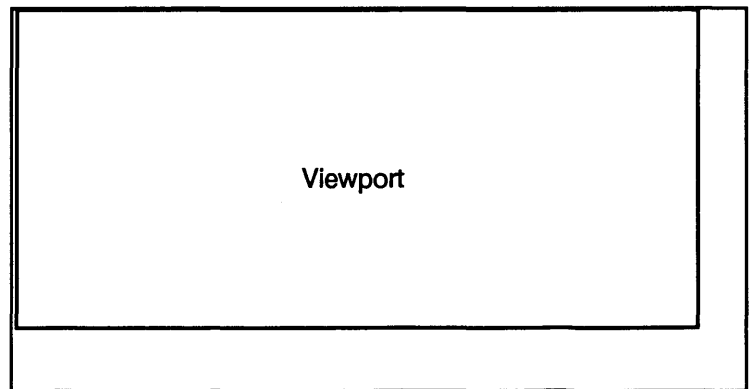
The statement looks like this:



To try a viewport now, type:

```
VIEWPORT 0,120,15,100,┘  
FRAME,┘
```

This puts in a viewport that extends along the x-axis from 0,0 to 0,120; and from 15 to 100 along the y-axis. The display now looks like this:

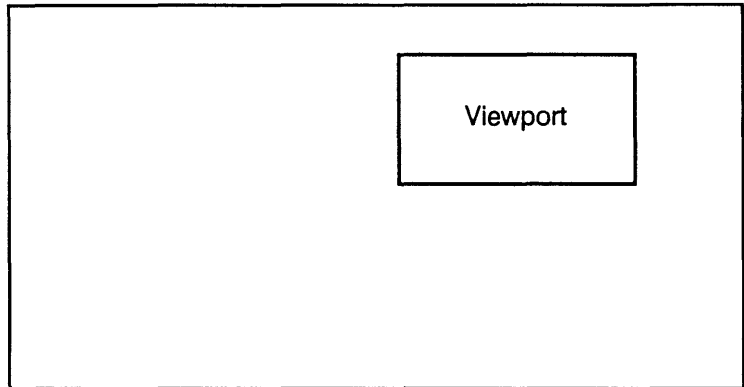


But that viewport is too large. Where will you put labels?

A Different VIEWPORT

Try a different, smaller viewport. Put it in the right of the viewing area:

```
GCLEAR_↵  
FRAME_↵  
VIEWPORT 60,90,60,80_↵  
FRAME_↵
```

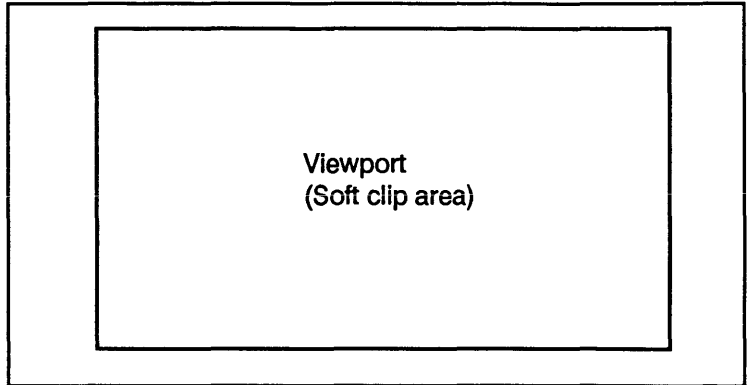


Now you have plenty of room for labels, all right. Maybe too much room:

Let's try again. Type:

```
GCLEAR_↵  
GINIT_↵  
FRAME_↵  
VIEWPORT 30,100,20,80_↵  
FRAME_↵
```

Voila! A nice-sized and centered plotting area, with plenty of room for labels.



Note



If you have a Vectra or other personal computer and the BASIC Language Processor, you may need to use `VIEWPORT` to make sure everything you draw is in the viewable area of the CRT. A good statement to use for most monitors, including an EGA, is `VIEWPORT 0,130,10,85`.

You can specify more than one viewport; but other graphics statements apply only to the last viewport specified.

The viewport is a *soft clip area*. You'll learn more about clipping later.

You now have a display area that's 70 GDU's by 60 GDU's in size. These values are in the computer's graphic display units; you can't change them.

But your data is in degrees (0-360) and in sine values (-1 to 1). How can you scale the plotting area?

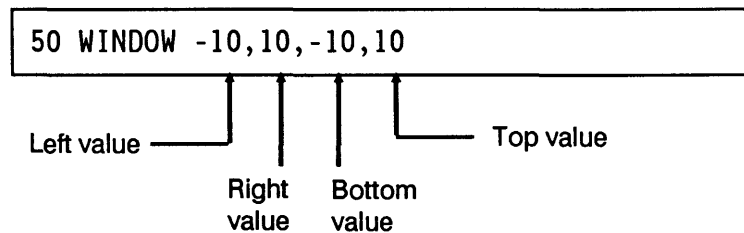
Simple: you use `WINDOW` or `SHOW`.

Scale the Plotting Area

Everything you've done so far has been in GDU's. Now it's time to specify your own units—call them "user-defined units," or UDU's—for the plotting area.

The WINDOW or SHOW statement assigns whatever values you specify to the boundaries of the viewing area defined by VIEWPORT.

Here's how:



This example makes a horizontal scale (x-axis) that extends from -10 to $+10$; and a vertical scale (y-axis) that's also -10 to $+10$.

You could use this for a semiconductor current/voltage curve, with current in milliamperes on the vertical and bias voltage on the horizontal scale.

SHOW or WINDOW defines the values which must be displayed within a viewport. Once you specify WINDOW or SHOW, you can forget GDU's: everything (well, almost) from here on assumes your own user-defined units of measure.

What's the difference between SHOW and WINDOW?

- **SHOW** forces *isotropic* units – UDU's that are an equal size in x- and y-directions. Even if you try to make unequal units, the statement forces them to be equal.
- **WINDOW** lets you specify units any way you want.

Try an example that demonstrates the difference between **SHOW** and **WINDOW**. First, type and run the mini-program below:

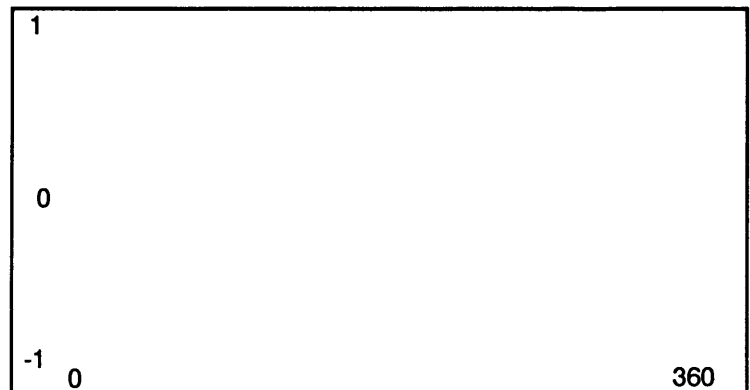
SCRATCH_␣

EDIT_␣

```
10 GCLEAR
20 GINIT
30 GRAPHICS ON
40 VIEWPORT 0,130,10,85
50 FRAME
60 WINDOW 0,360,-1,1
70 END
```

RUN_␣

The **WINDOW** statement in line 60 scales the viewport area into a plotting area that extends from 0 to 360 along the x-axis, and -1 to 1 along the y-axis.



This just happens to be a good scale for plotting the Trigator's data.

Now draw a line to a few typical points of the Trigator's sine data.

The sine of 90 degrees is 1, so type:

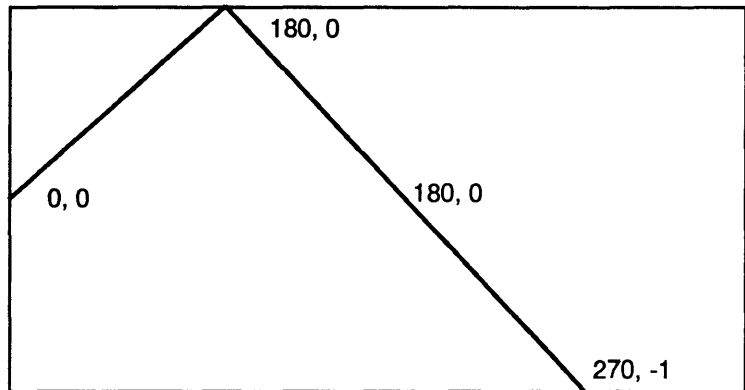
```
DRAW 90,1↵
```

The sine of 180 degrees is 0, so type:

```
DRAW 180,0↵
```

And the sine of 270 degrees is -1, so type:

```
DRAW 270,-1↵
```



You can see the values are scaled nicely on your screen.

Now change line 60 to use a SHOW statement:

```
EDIT 60↵
```

```
50 SHOW 0,360,-1,1↵
```



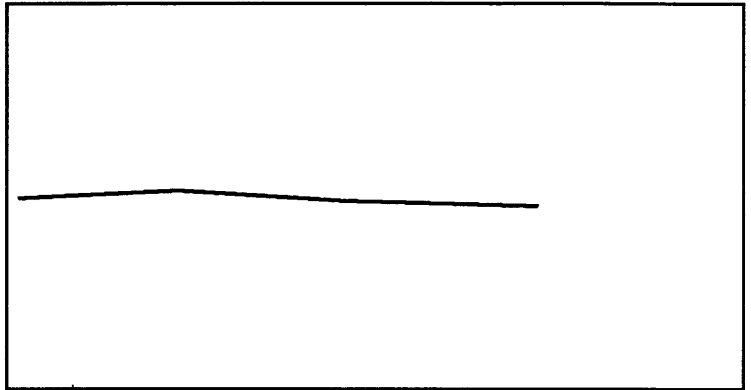
```
RUN ↵
```

Try drawing the same sine data again. Type:

```
DRAW 90,1 ↵
```

```
DRAW 180,0 ↵
```

```
DRAW 270,-1 ↵
```



See what happened? The `SHOW` statement ignored your specification of -1 to 1 for y , and instead scaled everything for isotropic (square) units, based on the x -axis specification.

What you have now, whether you like it or not, is a drawing area that's 360×360 units. It's the same as if you'd used this statement:

```
60 SHOW 0,360,-180,180
```

Cardinal Rule



SHOW always scales your drawing area with isotropic (square) units.

Which statement to use for the Trigolator plot?

- You know your x-axis data will be from 0 to 360.
- You know your y-axis data will be in the range -1 to $+1$.

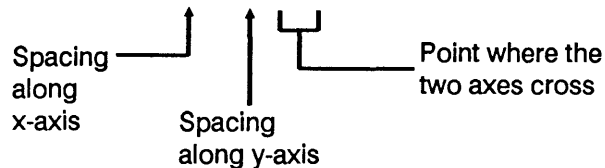
So it doesn't make sense to use isotropic (equal) units here. You'll use this:

```
230 WINDOW 0,360,-1,1
```

Put in Axes Lines

To help make your plot or graph meaningful, you can use axes lines with tic marks along them. You use the AXES statement to create tic marks. Here's how it's used:

```
100 AXES 45,.1,0,0
```



Try the following examples. First, type and run this mini-program:

```
SCRATCH_┘  
EDIT_┘
```

```
10 GCLEAR
20 GINIT
30 FRAME
40 VIEWPORT 20, 100, 10, 80
50 FRAME
60 WINDOW -10,10,-10,10
70 END
```

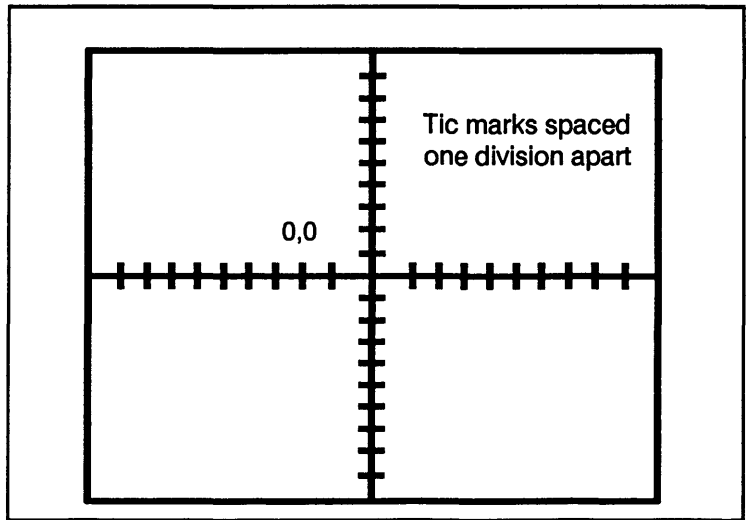
RUN ↵

The WINDOW statement in line 30 scaled the graphics area into an area -10 to $+10$ divisions in x , and -10 to $+10$ divisions in y .

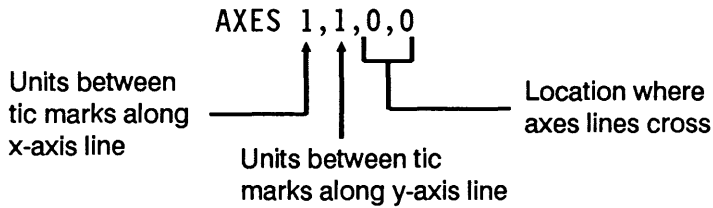
Now type:

AXES 1,1,0,0 ↵

This statement adds axes lines with tic marks spaced one division apart. The axes lines cross at point $0,0$.



The AXES statement puts in tic marks according to this scheme:

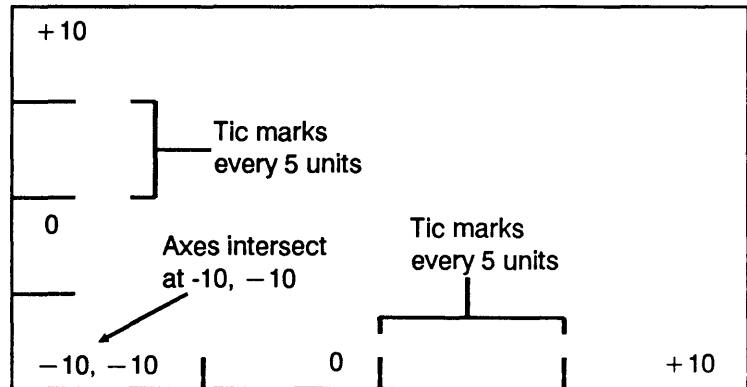


These are tic marks every unit along the x- and y-axis lines. The lines cross at 0,0.

Try moving the axes lines to the edges of the window. Type:

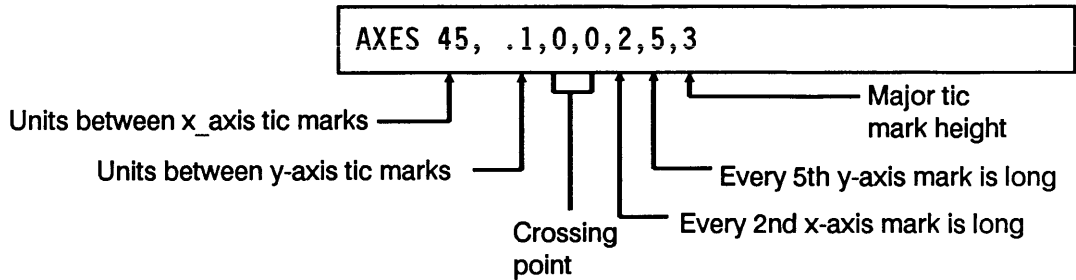
```
RUN_
AXES 5, 5, -10, -10_
```

Now the tic marks are spaced 5 units apart; and the axes intersect at $-10, -10$ — that is, in the lower left corner of the window.



More AXES Control

You've seen a simple AXES statement. But this statement also lets you put in axes lines with small and large tic marks. Here's how:

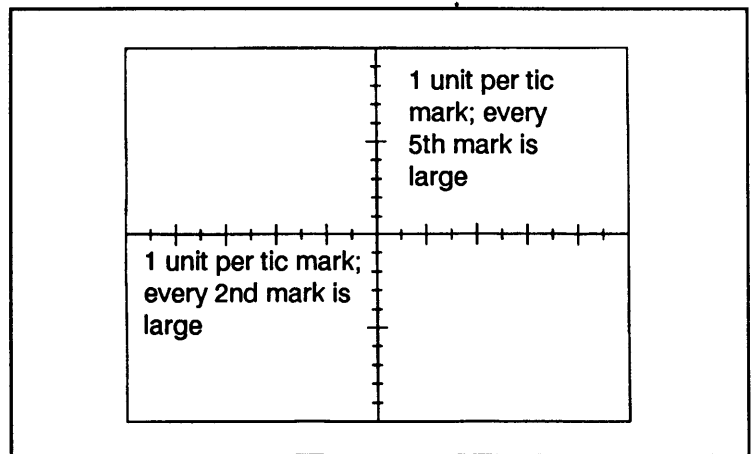


Try this one to put tic marks on your mini-program window:

```
RUN_
AXES 1,1,0,0,2,5,3_
```

This gives x- and y-axes with 1 unit between tic marks. Along the x-axis, every 2nd tic mark is a *major* mark – it's longer.

Along the y-axis, every 5th mark is long.



The length of ordinary tic marks is 1 GDU. The length of the longer, major marks is 3 GDU's. (Unlike the other numbers in the statement, the size of tic marks is *always* in GDU's.)

Plotting the Trigolator Axes

What tic marks to use for the Trigolator plot? Remember, the window is:

```
WINDOW 0,360,-1,1
```

So use an AXES statement that puts tic marks every 45 units along x, and every .1 unit along y, with the axes crossing in the center:

```
270 AXES 45, .1, 0, 0
```

Put in a Grid

Another way you can make your data more meaningful is to draw a grid over the plotting area.

You specify the grid in much the same way as you do axes; except you use the GRID statement, like this:

```
220 GRID 45, .1, 0, 0
```

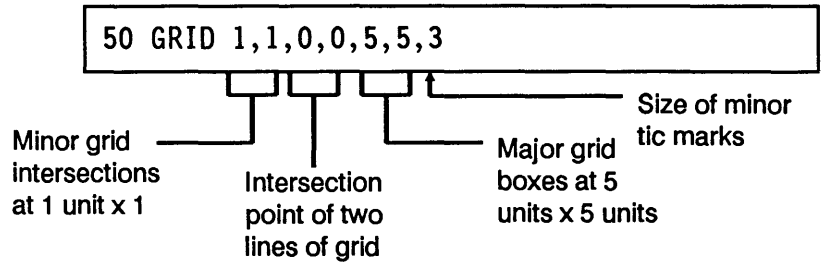
Units between
grid marks
along x_axis

Units between
grid marks along
y-axis

Intersection point of
two lines of grid

This is a perfect grid for the Trigolator plot, of course: each "box" of the grid is 45 x-axis units by 0.1 y-axis unit. One set of grid lines crosses at coordinates 0,0, and all other crossings are based on that one.

As with AXES, you can specify major "boxes" and minor crossings within the grid, like this:



This draws a grid with minor intersections at 1 unit x 1 unit; and major "boxes" of 5 units x 5 units. The "5,5" portion of this expression actually specifies the number of minor tic intervals between major grid lines.

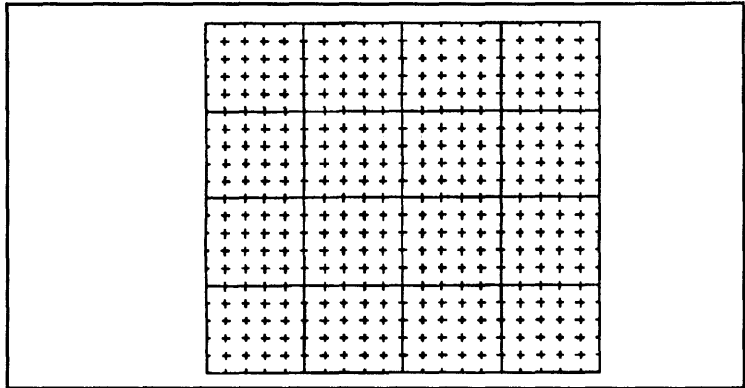
To see how it works, type:

```
RUN_
```

(Remember, this gives you a window using a WINDOW -10,10,-10,10 statement.) Now type:

```
GRID 1, 1, 0, 0, 5, 5, 3_
```

The result shows how this full-featured GRID statement lets you be very specific in plotting.



For the Trigolator plot, you'll use this statement:

```
310 GRID 45, .1, 0, 0, 4, 5, 3
```

It gives you a grid with:

- Small intersections at 45 x .1.
- Large boxes every 4 minor tic marks x 5 minor tic marks – that is, at 180 x .5.
- Major lines that are 3 GDU's long.

Label Your Plot

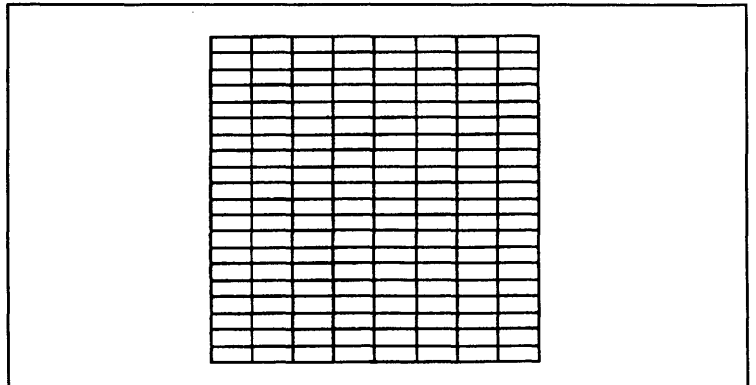
You're almost ready to plot some data! You have a viewport, a window, axes, lines and a grid. But what do they mean? You need to add some labels so that the world will know this is a Trigolator plot, and not merely a drawing of a checkerboard or a chain-link fence.

First, write a mini-program that sets up most of your Trigolator plot so far. Type:

```
SCRATCH_↵  
EDIT_↵
```

```
10 GCLEAR  
20 GINIT  
30 FRAME  
40 VIEWPORT 30,100,20,80  
50 FRAME  
60 WINDOW 0,360,-1,1  
70 GRID 45,.1,0,0  
80 END
```

```
RUN_↵
```



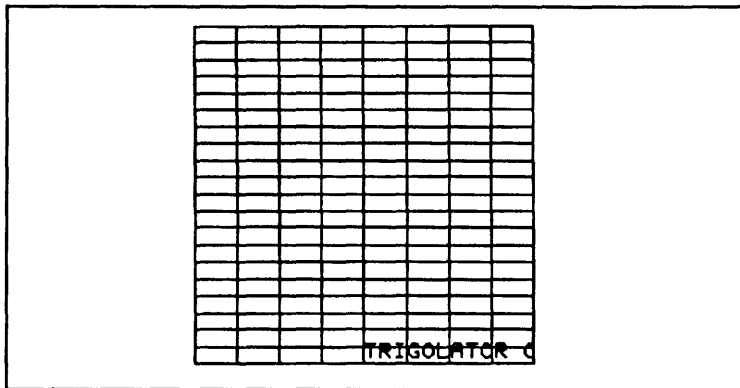
You see the viewport and the grid you'll use for the Trigatorator output plot.

The LABEL statement lets you put labels in your graphs and plots. To put the label TRIGOLATOR OUTPUT at the bottom of the plotting area, move the pen to where you want the label. Then use the LABEL statement.

Try it. Type:

```
MOVE 180, -1 ↵  
LABEL "TRIGOLATOR OUTPUT"
```

This moves the pen to the bottom of the plot area. But what you see probably isn't where you want the label, or how you want it to look. What's wrong with this picture?



The problem is *clipping*.

Clipping and Unclipping

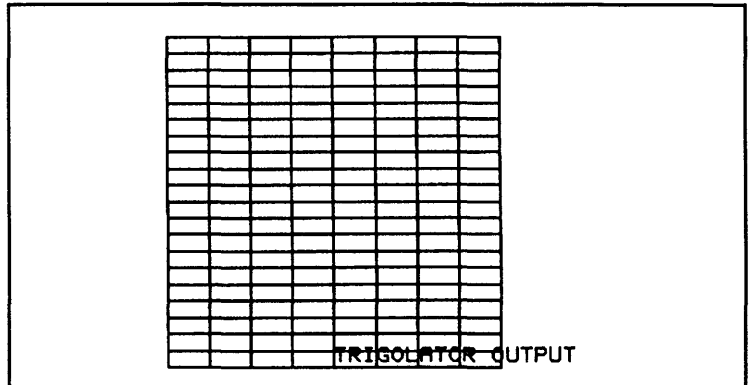
Remember, the plot area (the viewport you specified earlier) is also called the "soft clip area." With clipping *on*, you can't plot anything outside the viewport limits.

You have to turn clipping off to allow plotting outside the viewport.

Try it this way; type:

```
CLIP OFF_↵  
MOVE 180, -1_↵  
LABEL "TRIGOLATOR OUTPUT"_↵
```

That's a little better. Now you can plot outside the viewport.



When clipping is on, you can't put labels or drawing outside the viewport. To turn clipping on, you can use:

```
CLIP ON_↵
```

When you need to put something outside the viewport (such as a label), use:

```
CLIP OFF
```

Remember:

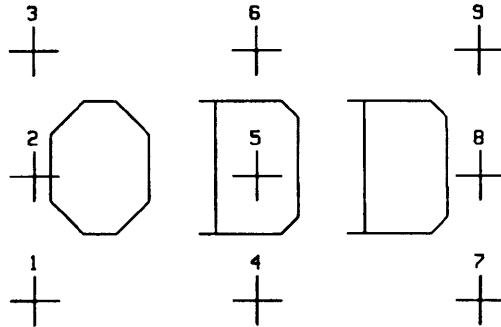
- To limit plotting to areas inside the viewport only, use CLIP ON.
- To allow plotting outside the viewport, use CLIP OFF first.

You can *never* plot or draw outside the graphics area. For this reason it's sometimes called the "hard clip area."

Using LORG

Your label is looking better. But it's still not *below* the viewport. There are a few techniques for putting it there, but the surest is with the LORG (label origin) statement.

LORG specifies the relative origin of labels with respect to the current pen position.



For example, in the case of the label ODD, the following statement centers the label below the current pen position (marked here by cross 6):

```
LORG 6  
LABEL "ODD"
```

Lines 100 and 110 below put the label above and to the right of the current pen position.

```
100 LORG 1  
110 LABEL "ODD"
```

Since you want the label to be centered beneath the current pen position, you'll use LORG 6.

So type:

```
RUN↵  
CLIP OFF↵  
MOVE 180, -1↵  
LORG 6↵  
LABEL "TRIGOLATOR OUTPUT"↵
```

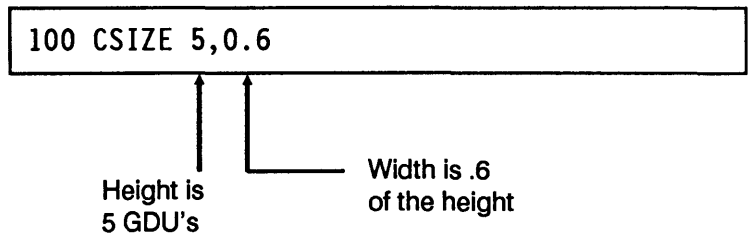
That's better. The label is right where it's supposed to be. But it's a little small, don't you think?

Using CSIZE

To change the size of a label, use the CSIZE (character size) statement. You can specify the size alone, like this:

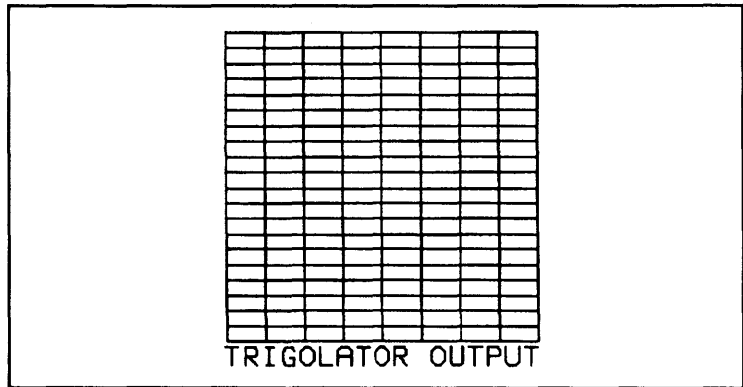
```
CSIZE 7
```

Or you can specify the size and aspect ratio (ratio of width to height), like this:



For the Trigolator plot label, try CSIZE 7. Type this:

```
RUN↵  
CLIP OFF↵  
MOVE 180,-1↵  
LORG 6↵  
CSIZE 7↵  
LABEL "TRIGOLATOR OUTPUT"↵
```



Not bad! Now add one more label, Trigovalue, along the left-hand side of the plot.

Using LDIR

The LDIR (label direction) statement lets you rotate a label. It looks like this:

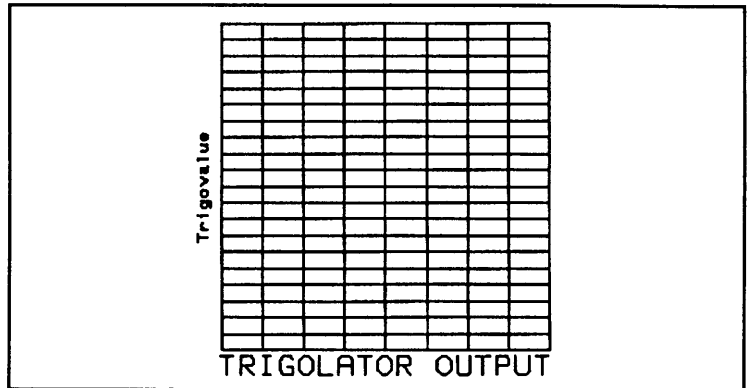
```
LDIR 120
```

This means "rotate the next labels 120 radians (or degrees)."

The angle is in the current angle unit (degrees or radians), and is interpreted as counterclockwise, from three o'clock.

To add another label, "Trigovalue," at the left of the plot, use LDIR 90 to rotate the label 90 degrees. The computer "wakes up" in radians mode, remember, so change to degrees mode with DEG. Type:

```
MOVE -30,0↵  
DEG↵  
LDIR 90↵  
CSIZE 4↵  
LABEL "Trigovalue"↵
```



The last LORG statement you used, LORG 6, is still active unless you change it.

This example shows something else: you can specify locations *outside* the viewport, using your own UDU's. In this case, typing MOVE -30,0 placed the pen 30 units to the left of 0,0 – that is, left of the viewport.

Plot Your Data

At last, you're ready to actually plot data. If you've followed all the previous steps – and were especially careful in using WINDOW or SHOW to scale the plotting area – you should have few problems.

Alas, no more models of the Turveydrop Trigolator exist. However, you can *simulate* the Trigolator's data by plotting the sines of angles from 0 to 360.

First, add lines 80-120 to your mini-program:

EDIT 80_␣

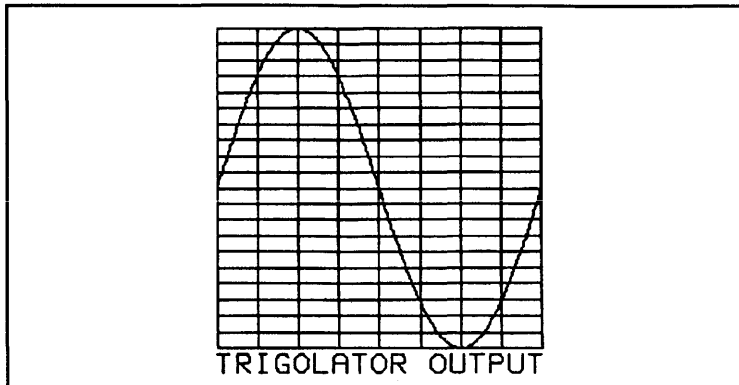
```
80 CLIP OFF
90 MOVE 180, -1
100 LORG 6
110 CSIZE 7
120 LABEL "TRIGOLATOR OUTPUT"
```

Now type the following to simulate data from the Trigolator:

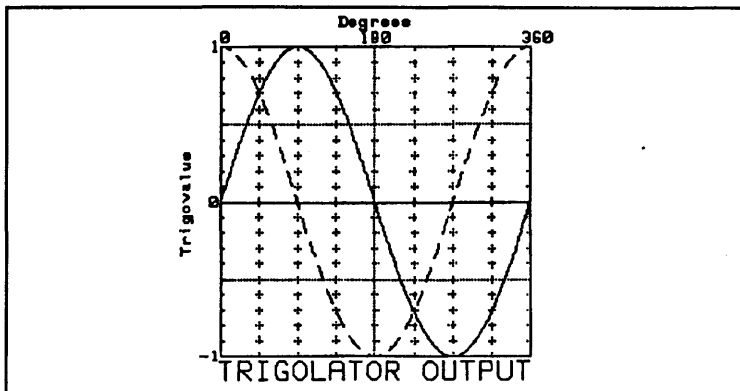
```
130 MOVE 0,0
140 DEG
150 FOR I = 0 TO 360
160 DRAW I, SIN(I)
170 NEXT I
180 END
```

Finally, run the mini-program to see the data plotted:

RUN_␣



If you have problems, or want to see a completed Trigator plot, load the program "TRIGOLATOR" from your disk of examples. That program draws both the sine and cosine of angles from 0 to 360 degrees.



It also uses an improved grid, along with additional labels. But it's essentially the same program you just finished writing. There's a listing on the next few pages.

```
10 !RE-STORE "TRIGOLATOR"
20 GOSUB Init
30 GOSUB Set_view
40 GOSUB Scale
50 GOSUB Place_axes
60 GOSUB Place_grid
70 GOSUB Set_labels
80 GOSUB Draw
90 STOP
100 Init: !
110 DEG
120 GRAPHICS ON
130 ALPHA OFF
140 GINIT
150 GCLEAR
160 FRAME
170 RETURN
180 Set_view: !
190 VIEWPORT 30,100,20,80
200 FRAME
210 RETURN
220 Scale: !
230 WINDOW 0,360,-1,1
240 RETURN
250 Place_axes: !
260 CLIP ON
270 AXES 45,.1,0,0,4,5,3
280 RETURN
290 Place_grid: !
300 LINE TYPE 4
310 GRID 45,.1,0,0,4,5,3
320 RETURN
```

```
330 Set_labels: !
340 CLIP OFF
350 LINE TYPE 1
360 MOVE 0,1
370 LORG 1
380 CSIZE 4
390 LABEL "0"
400 MOVE 350,1
410 LABEL "360"
420 MOVE 0,0
430 LORG 8
440 LABEL "0"
450 MOVE 0,1
460 LABEL "1"
470 MOVE 0,-1
480 LABEL "-1"
490 MOVE 180,1
500 LORG 4
510 LABEL "180"
520 MOVE 180,1.1
530 LABEL "Degrees"
540 MOVE -30,0
550 LDIR 90
560 LABEL "Trigovalue"
570 LDIR 0
580 MOVE 180,-1
590 LORG 6
600 CSIZE 7
610 LABEL "TRIGOLATOR OUTPUT"
620 RETURN
630 Draw: !
640 MOVE 0,0
650 FOR I=0 TO 360
660 DRAW I, SIN (I)
670 NEXT I
680 MOVE 0,0
690 LINE TYPE 5
700 FOR I=1 TO 360
710 DRAW I, COS(I)
720 NEXT I
730 RETURN
740 END
```

Self-Computing the Scale

Sometimes you don't know the limits of the data, so it's difficult to specify parameters for WINDOW, SHOW, GRID, etc.

If this happens to you, don't despair. The following section of code shows how a plot subroutine used with the HP 8980A Vector Analyzer automatically scales the plot area to the correct user units. (You'll find more details about the HP 8980A in back in lesson 18.)

```
350 Plot: !
360 ALPHA OFF
370 I_max=MAX(I(*))
380 I_min=MIN(I(*))
390 I_max=MAX(ABS(I_max),ABS(I_min))
400 Q_max=MAX(Q(*))
410 Q_min=MIN(Q(*))
420 Q_max=MAX(ABS(Q_max),ABS(Q_min))
430 Max=MAX(1.2*I_max,1.2*Q_max)
440 !
450 GINIT
460 GRAPHICS ON
470 VIEWPORT 30,105,15,90
480 CLIP OFF
490 MOVE 75,0
500 DEG
510 LDIR 0
520 LORG 6
530 LABEL "I AXIS"
540 MOVE 0,52
550 LDIR 90
560 LORG 4
570 LABEL "Q AXIS"
580 CLIP ON
590 SHOW -Max,Max,-Max,Max
600 GRID Max/5,Max/5,0,0
```

```
610 AXES Max/25,Max/25,0,0,5,5
620 FRAME
630 PENUP
640 FOR M=0 TO Num_pairs-1
650 I_value=I(M)
660 Q_value=Q(M)
670 PLOT I_value,Q_value,1
680 NEXT M
690 !
700 PENUP
710 DISP ""
720 RETURN
```

How it works: When this subroutine is called, the data points are in two arrays, I and Q. There's also a variable, Num_pairs, that contains the total number of data pairs.

Line 370 finds the maximum value of I, and line 380 finds its minimum (which may be below zero).

Line 390 then uses I_max and I_min to determine the largest existing value for I in the array.

Lines 400-420 find the largest value for Q in the same way.

Then line 430 determines the *maximum* limits. By multiplying I_max and Q_max by 1.2, the program guarantees a comfortable 20% margin at the edges of the plotted data.

Once the maximum value (Max) is determined, it's then used in the SHOW, GRID, and AXES statements (lines 590-610) to specify the plotting area units, and grid and axes lines.

Dump to a Printer

To get a hard-copy output of your plot, you can dump graphics to any printer, including the HP LaserJet, that conforms to the HP Raster Interface Standard.

You use the DUMP GRAPHICS and DUMP DEVICE IS statements:

```
100 DUMP DEVICE IS 26
110 DUMP GRAPHICS
```

Line 100 specifies a printer at address 26 as the device to which graphics will be dumped. Then line 110 dumps the entire graphics area, which is printed.

You also can put the address of the printer right in the DUMP GRAPHICS statement (just like LIST and CAT):

```
100 DUMP GRAPHICS #26
```

There's a Lot More...

As you may suspect, you've barely scratched the surface of HP BASIC's graphics capabilities. To learn about all the myriad things you can do with graphics, you'll want to refer to other HP manuals that deal with this subject in more detail.

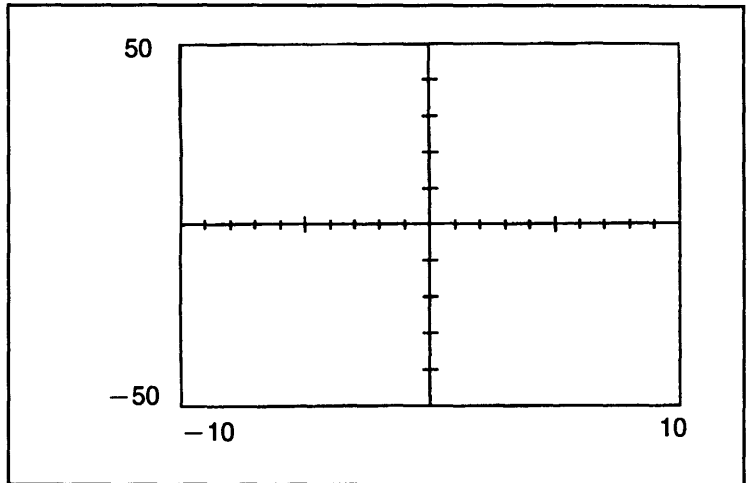
No matter how elegant and technically sophisticated your programs are, nobody sees your code. All that's visible to the outside world is how your data is displayed. That's why good graphics are so important.

Review Quiz

1. Following this WINDOW statement:

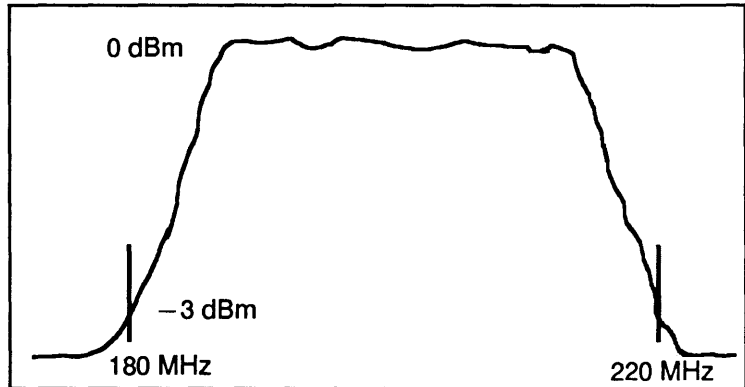
```
WINDOW -10,10,-50,50
```

What statement would you use to draw the axes lines shown here?



2. Which statement always scales your viewport for equal-sized units vertically and horizontally?

3. Clara Peggotty is using a spectrum analyzer to monitor the signal passed through a filter. With a constant-amplitude input, the filter gives an output between 180 MHz and 220 MHz that looks like this:



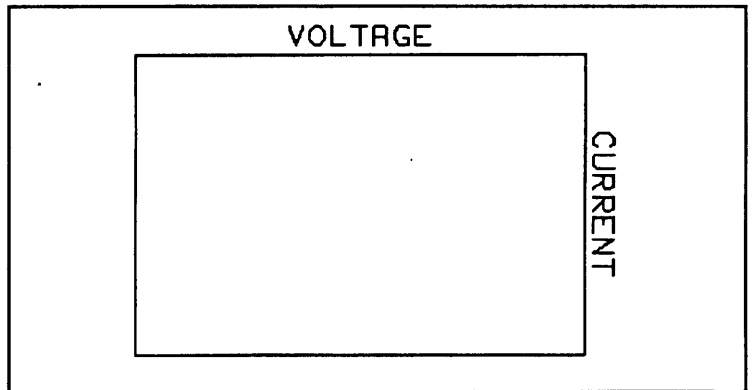
Write a WINDOW statement that scales the plotting area to allow this curve to be shown.

All that Peggotty cares about is the area between 180 and 220 MHz.

4. The following code establishes a viewport and a plotting area within the viewport.

```
10 GCLEAR
20 GINIT
30 FRAME
40 VIEWPORT 20,110,20,80
50 WINDOW 0,10,0,20
60 FRAME
70 CSIZE 7
```

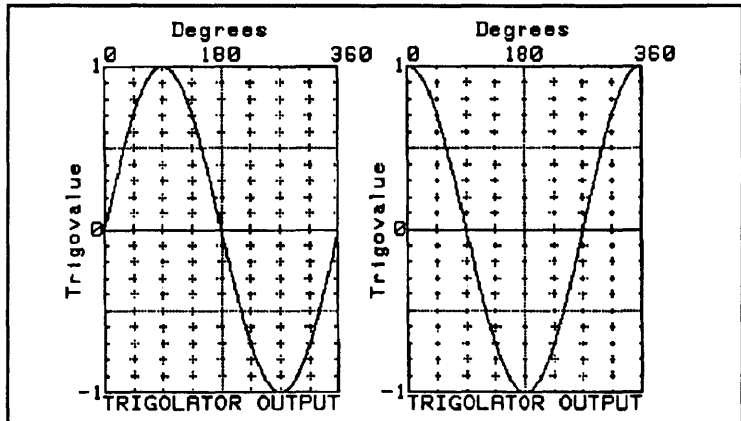
Write additional code to add labels as shown here:



(Don't forget the trigonometric mode!)

Laboratory Exercise

Modify the program "TRIGOLATOR" to provide *two* plotting areas; and plot the sine curve in one area, the cosine curve in the other. Your output should look like this:



One solution is shown as SOL_LAB19 on your disk of examples.

How to Design a Complete Program

You've come to the end of part 2. If you worked through lessons 11-19 (and perhaps part 1, lessons 1-10 as well), you should now have a pretty good idea of the programming techniques needed for plain-vanilla BASIC instrument control.

There's just one thing left – writing a complete program. And as you begin to write more complex "real-world" programs, you'll find a structured approach usually gets you to the solution more quickly and easily.

In this lesson you'll learn a structured approach to programming for instrument control. You'll learn about:

- A step-by-step procedure for writing a program.
- Warnier-Orr diagrams.
- Pseudocode.
- Modularizing code into small segments.
- Debugging aids and hints.

Writing a Structured Program

Now, there are a lot of ways to write a program. This lesson shows you a structured procedure. It has the advantage of *modularizing* your program—breaking it down into simple tasks for easier testing, debugging, and documentation.

And it'll give you a program that's easier for other people to understand.

There are a number of "tools" you can use to build your program: flowcharts, diagrams, pseudocode, etc. You may find some of these helpful, others not.

Here are the general steps you'll follow to create the program in this lesson:

1. Determine how you'd make the measurement manually.
2. Check out the hardware setup.
3. Use structured programming techniques and pseudocode to create a Warnier-Orr diagram.
4. Determine subroutines and subprograms.
5. Write the program code.
6. Test and debug.
7. Document the program.

Follow these steps as you create a solution for the problem below.

The problem: Your task is to determine the frequency response of a filter over the range from 1 kHz to 10 kHz. You have:

- The filter.
- HP 3456A Digital Voltmeter.
- HP 3325B Function Generator.

This Lesson's Featured Instrument: The HP 3325B Synthesizer/Function Generator

This high-performance instrument includes a synthesizer with 11-digit resolution, a function generator with precision waveforms, a wideband sweeper, and full HP-IB capability.

Frequency range for a sine wave is 0.000001 Hz to 21 MHz (to 11 MHz for square and triangle waves). Resolution is 1 mHz above 100 kHz, and 1 μ Hz below 100 kHz. At 1 V p-p output, sine wave amplitude accuracy is ± 0.4 dB in the 100 kHz to 20 MHz range.

HP-IB Capabilities

HP-IB interface capabilities are: SH1, AH1, T6, L3, SR1, RL1, PP0, DC1, DT0, C0, E1. The default HP-IB address is 17.

The HP-IB instrument command strings you'll use in this lesson are listed as comments right in the program "FILTER_TST."

Another Featured Instrument: The HP 3456A Digital Multimeter

This Digital Multimeter (often called a digital voltmeter, or DVM) is a microprocessor-based, fully guarded integrating instrument for measuring dc and true rms ac voltage, as well as resistance.

Resolution on the 1 V ac scale is 1 μ V, with an input impedance of 1 megohm.

HP-IB Capabilities

The instrument's HP-IB capability codes are: SH1, AH1, T5, L4, SR1, RL1, PP0, DC1, DT1, C0, E1. Default address on HP-IB is 22. The HP 3456A has an SRQ button on its front panel that can be used to signal or interrupt the controller.

As with the HP 3325B, you'll find descriptions of the HP 3456A's instrument command strings included as remarks in this lesson's program.

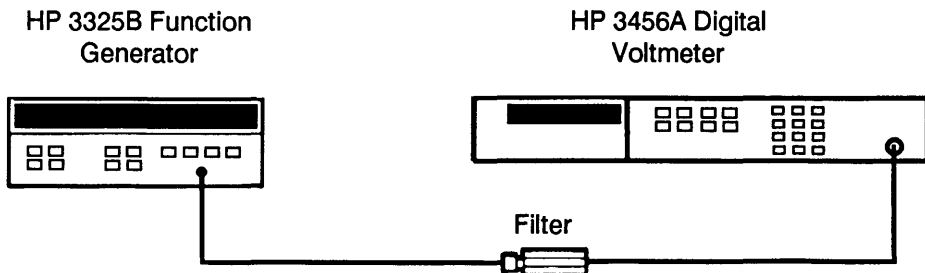
Determine a Manual Solution

To find out the frequency response of the filter, you'd connect it as shown below.

Then you'd follow this procedure to test the filter:

1. Set the function generator to 1 kHz, sine wave output, 1 V rms.
2. Read and record the value of input to the digital voltmeter.
3. Set the function generator to the next frequency—say, 1.1 kHz.
4. Read and record the value of input to the DVM.
5. Continue with steps 3 and 4 until you've reached 10 kHz.
6. Plot the data on graph paper. (Frequency response is a lot easier to read if it's graphed!)

True, this procedure would probably take most of a day—or two. But it shows how you'd make the measurements. And lucky for you, you're going to do the same thing with HP BASIC and HP-IB, and let the computer do the grunt work.



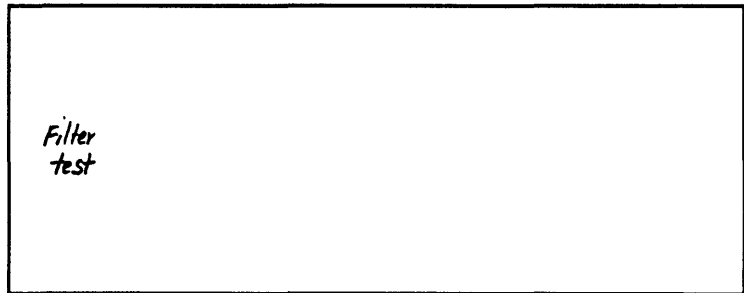
Check Out Hardware

Your next step is to make the hardware setup. Attach HP-IB cables to the instruments, determine their addresses, and test each instrument with the REMOTE and LOCAL statements to see if you have control.

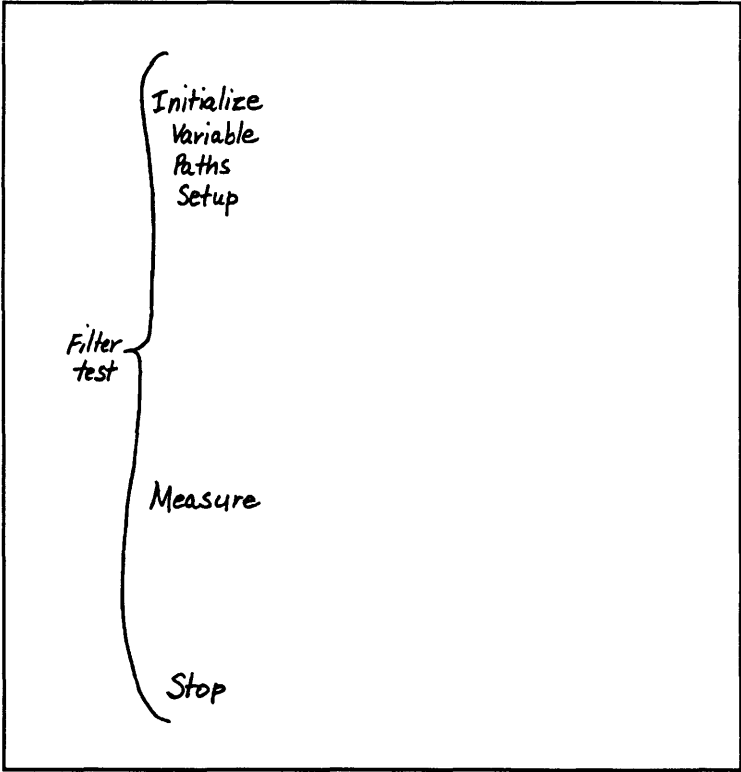
Create a Warnier-Orr Diagram

Next you draw up a Warnier-Orr diagram for the procedure. The Warnier-Orr diagram is used to break down a task into smaller tasks. Then it breaks down these tasks into smaller ones yet.

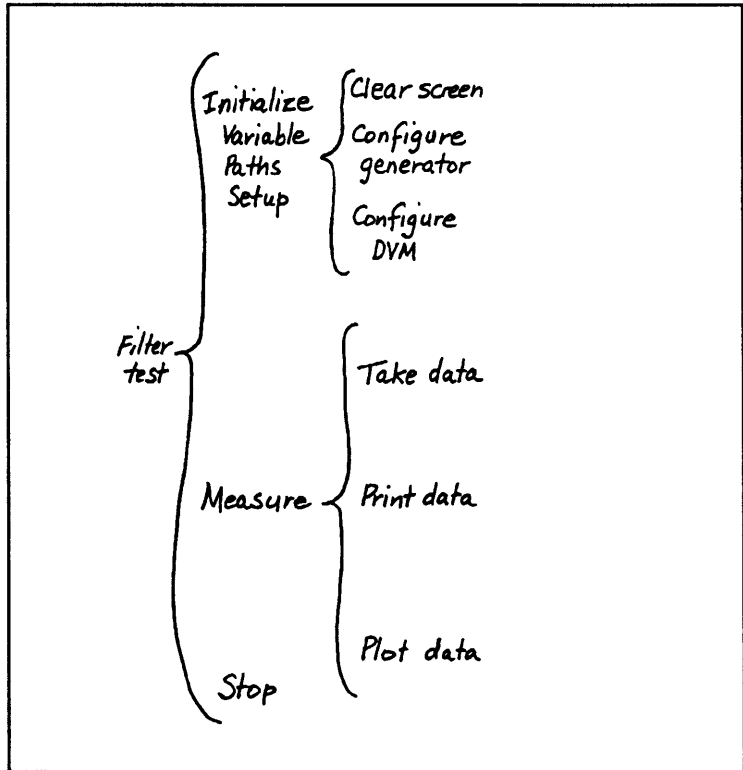
You start off by writing the main task on the left-hand side of a big piece of paper:



Then you decide what primary functions you'll need to do. These will eventually become subprograms, or different parts of a main program.

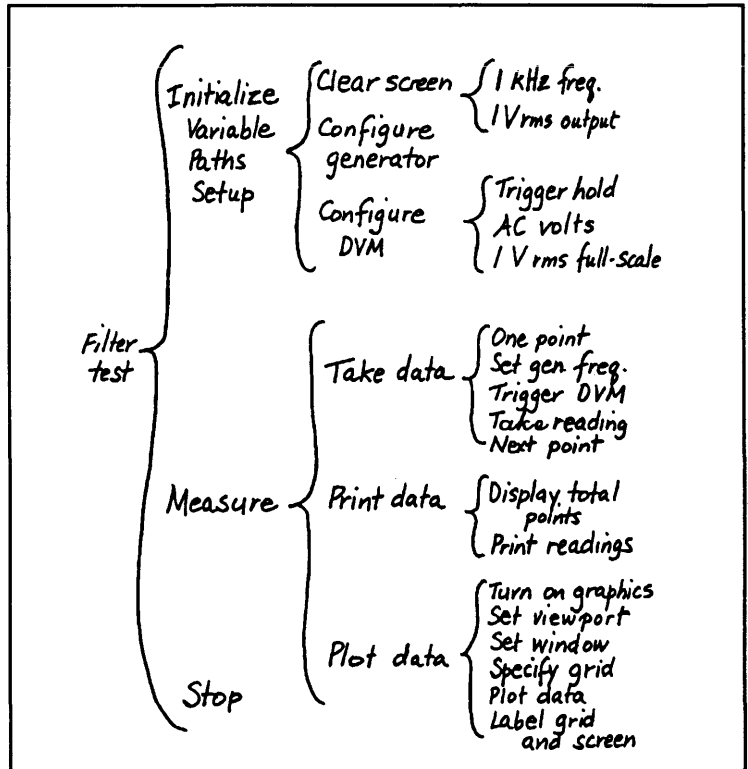


Keep subdividing tasks into smaller ones as you go from left to right:



This level will be the subroutines in your program.

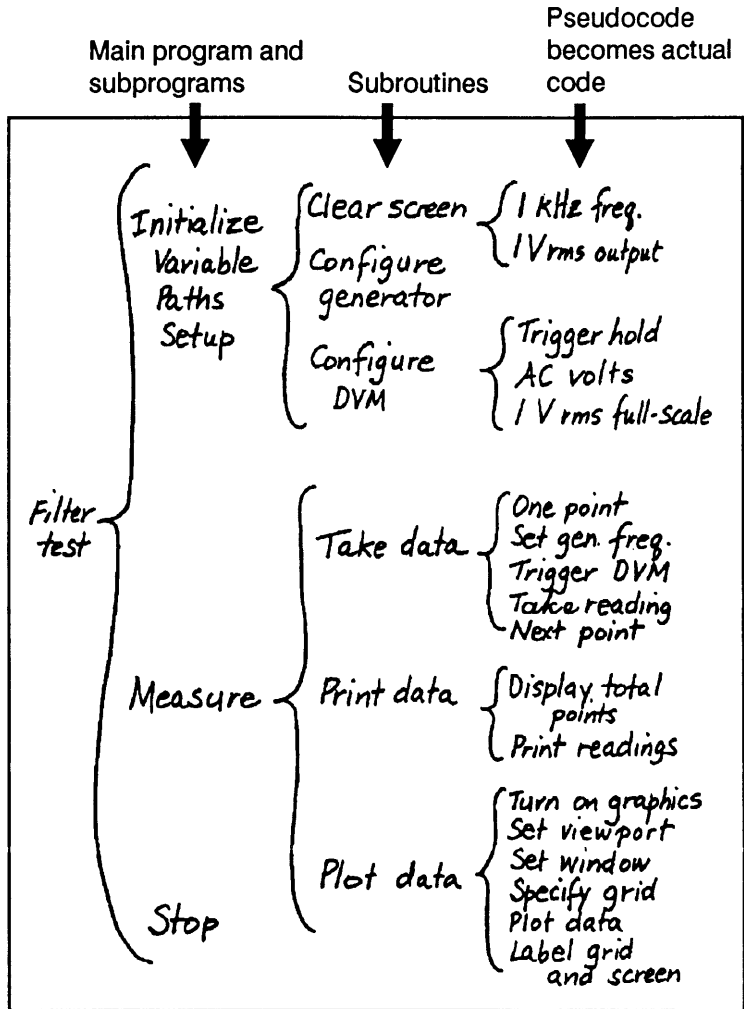
You can write pseudocode – a kind of shorthand – that outlines exactly what you'll do to complete each task:



Ideally, you should know all your variables now, so you can initialize them. But to be honest, even hotshot programmers have to come back to the Initialize pseudocode and add variables later. So leave yourself a little room here.

Determine Subprograms and Subroutines

This is the easy part, if you've drawn a good Warnier-Orr diagram. The tasks on the left become your main program (or subprograms, if the program is very large). Tasks in the middle are subroutines.



You can use the pseudocode on the right to create the actual code.

Write the Program Code

Now you can write the actual code for the program. Write the code on paper first, then enter it into the computer later.

You can make your subprograms and subroutines the same names as on the Warnier-Orr diagram.

This is also the time when you'll want to look at the instrument manuals to decide what instrument command strings to OUTPUT.

Once you've written the code on paper, type it into the computer. As you do, "comment out" all subprogram and subroutine calls (with REM or an exclamation point), so you can test and debug each task separately later.

The program to test the filter is on your disk of examples as "FILTER_TST". There's a listing of the program on the next few pages.

```

10 ! RE-STORE "FILTER_TST"
20 !
30 ! LAST REVISED: August 15, 1988
40 !
50 ! This program measures the frequency
60 ! response of a filter. The output from
70 ! a function generator is applied to the filter
80 ! and stepped through a range of 1 kHz
90 ! to 10 kHz. Output from the filter is
100 ! fed to a digital voltmeter, and a reading
110 ! on the DVM is taken at each frequency step.
120 !
130 ! Equipment required:
140 ! HP3456A Digital Voltmeter at 722
150 ! HP3325B Function Generator at 717
160 !
170 !
180 ! XXXXXXXXXXXX MAIN PROGRAM XXXXXXXXXXXX
190 !
200 Variables: !
210 OPTION BASE 1
220 INTEGER First,Freq,Incr,Last,Nr,Size
230 First=1000
240 Last=10000
250 Incr=100
260 Size=(Last-First)/INCR)+1
270 ALLOCATE REAL Reading (Size)
280 ALLOCATE Cmd$[80]
290 ! First = First frequency in the range
300 ! Freq = Current frequency setting
310 ! Incr = Frequency increment size
320 ! Last = Last frequency in the range
330 ! Nr = Number of the current reading
340 ! Size = Size of the array 'Readings',
350 !           and the total number of readings.
360 ! Readings = Array to hold all readings
370 ! Cmd$ = Temporary command string
380 !

```

```
390 ! ++++++
400 !
410 Paths: !
420 ASSIGN @Crt TO 1
430 ASSIGN @Kbd TO 2
440 ASSIGN @Gen TO 717
450 ASSIGN @Dvm TO 722
460 !
470 !
480 ! ++++++
490 !
500 Setup: !
510 GOSUB Clear_crt
520 GOSUB Config_gen
530 GOSUB Config_dvm
540 !
550 !
560 ! ++++++
570 !
580 Measure: !
590 GOSUB Take_data
600 GOSUB Print_data
610 PAUSE
620 DISP "PRESS [CONTINUE] TO PLOT DATA"
630 GOSUB Clear_crt
640 GOSUB Plot_data
650 ++++++
660 Stop: !
670 DISP "PROGRAM FINISHED"
680 STOP
690 !
700 !
```

```
710 ! XXXXXXXXXXX SUBROUTINES XXXXXXXXXXX
720 !
730 Clear_crt: !
740 CLEAR SCREEN
750 RETURN
760 !
770 ! ++++++
780 !
790 Config_gen: !
800 DISP "CONFIGURING GENERATOR"
810 CLEAR @Gen
820 WAIT 2
830 OUTPUT @Gen;"FU1 AM1VR RF1"
840 DISP "
850 RETURN
860 !
870 ! CLEAR @Gen resets a baseline state
880 ! FU1 = Set Function 1, Sine output
890 ! AM1VR = Amplitude is 1 Volt RMS
900 ! RF1 = Use Front panel output jack
910 ! Note: The 3325 ignores spaces
920 !
930 ! Some other conditions set by 'CLEAR':
940 ! FR1KH = Frequency 1 KHz (to be set below)
950 ! OFOVO = No offset voltage
960 ! Single frequency mode - not sweeping
970 !
980 !
```



```
990 ! ++++++
1000 !
1010 Config_dvm: !
1020 DISP "CONFIGURING VOLTMETER"
1030 CLEAR @Dvm
1040 WAIT 2
1050 Cmd$="T4 F2 R3 Z0 1STI"
1060 OUTPUT @Dvm;Cmd$
1070 DISP "          "
1080 RETURN
1090 !
1100 ! CLEAR @Dvm resets a baseline state
1110 ! T4 = Trigger hold, to stop readings
1120 ! F2 = Function 2, AC Volts
1130 ! R3 = Range 3, 1 Volt RMS full scale
1140 ! Z0 = Auto Zero off, for higher speed
1150 ! 1STI = Integrate over 1 Power Line Cycle
1160 ! Note: The 3456 ignores spaces
1170 !
1180 ! Some other conditions set by 'CLEAR':
1190 ! M0 = Math off
1200 ! 1STN = 1 reading per trigger
1210 ! FLO = Filter off
1220 ! RSO = Reading Storage off
1230 ! S01 = System output on
1240 ! Default delay for ACV is 60 ms
1250 !
1260 !
```

```

1270 ! ++++++
1280 !
1290 Take_data: !
1300 DISP "TAKING DATA ON THE FILTER"
1310 Nr=0
1320 FOR Freq=First TO Last STEP Incr
1330 Nr=Nr+1
1340 Cmd$="FR"&VAL$ (Freq) & "HZ"
1350 OUTPUT @Gen;Cmd$
1360 OUTPUT @Dvm;"T3"
1370 ENTER @Dvm;Reading(Nr)
1380 NEXT Freq
1390 DISP "      "
1400 RETURN
1410 !
1420 ! If Freq=1000, then Cmd$="FR1000HZ"
1430 ! VAL$ ( ) converts a numeric value into a
1440 ! string of characters that represent it.
1450 ! This is one way of getting the value for
1460 ! frequency into the instrument.
1470 ! Here are some equivalent techniques:
1480 !
1490 ! 10 Fmt1: IMAGE "FR",K,"HZ"
1500 ! 20 OUTPUT @Gen USING Fmt1;Freq
1510 ! or
1520 ! 10 OUTPUT @Gen;"FR";Freq;"HZ"
1530 !
1540 ! 'T3' triggers the DVM for a single
1550 ! reading, which is 'read' by ENTER
1560 !
1570 !

```

```

1580 ! ++++++
1590 !
1600 Print_data: !
1610 DISP "FILTER DATA, TOTAL POINTS = ";Size
1620 PRINT Reading (*)
1630 WAIT 2
1640 RETURN
1650 !
1660 !
1670 ! ++++++
1680 !
1690 !
1700 Plot_data: !
1710 GINIT !Initialize graphics
1720 GRAPHICS ON !Turn on the graphics screen
1730 PEN 1 !Select a pen
1740 VIEWPORT 15,120,30,95 ! Specify a viewport
1750 Min_yaxis=0
1760 Max_yaxis=.5 !Filter output can't exceed .5V or it won't plot
1770 WINDOW First,Last,Min_yaxis ! Set the window for:
1780 ! Minimum x = first frequency
1790 ! Maximum x = last frequency
1800 ! Minimum y = 0 volts
1810 ! Maximum y = maximum y-axis volts
1820 CLIP ON
1830 GRID Incr*10,(Max_yaxis-Min_yaxis)/10,Min_yaxis,First !Plot a grid
1840 MOVE First,Reading(1) !Move to the first point to be plotted
1850 !
1860 ! Plot the data
1870 !
1880 FOR I=1 TO Size
1890 DRAW First+(I-1)*Incr,Reading(I)
1900 NEXT I
1910 !

```

```

1920 ! Label the x-axis grid divisions
1930 !
1940 DEG
1950 LDIR 0
1960 LORG 6
1970 CLIP OFF
1980 FOR X=First TO Last STEP Incr*10
1990 MOVE X,Min_yaxis
2000 LABEL USING "SDD";X/1000
2010 NEXT X
2020 !
2030 ! Label every second y-axis grid division
2040 !
2050 LORG 8
2060 FOR Y=Min_yaxis TO Max_yaxis STEP (Max_yaxis-Min_yaxis)/5
2070 MOVE First,Y
2080 LABEL Y
2090 NEXT Y
2100 !
2110 ! Change the viewport to the entire screen
2120 !
2130 VIEWPORT 1,100*RATIO,1,100
2140 WINDOW 1,100*RATIO,1,100
2150 !
2160 ! Put a nice label at the bottom of the plot
2170 !
2180 ! MOVE 65,20 ! Absolute screen position
2190 LDIR 0
2200 LORG 5
2210 LABEL "FREQ (kHz)"
2220 !
2230 ! Put a nice label on the y-axis
2240!
2250 LORG 5
2260 LDIR 90

```

```
2270 MOVE 3,60 ! Absolute screen position
2280 LABEL "VOLTS RMS"
2290 !
2300 RETURN
2310 !
2320 ! ++++++
2330 End: !
2340 END
```

Test and Debug



It's time to run the program...and inevitably, that means testing and "debugging" it too.

"Commenting out" the subprograms and subroutines makes this part of the process easier:

First, run the program without any subroutine or subprogram calls. (That is, leave them all commented out.) Fix any errors you find.

Next, delete the REM or exclamation point in front of the first subprogram or subroutine call (Init_var in the example) and run the program again. Fix any new errors.

Continue this way, stripping out remarks and running and fixing the program, until everything runs correctly.

Debugging Aids

Debugging is often more of an art than a science. However, HP BASIC and your computer have some features that can help in your search for those elusive little critters known as bugs.

- *Pre-run*: Pre-run checks for proper program structure before the program actually runs.

- **[STEP]:** The [STEP] key lets you execute a line at a time to determine program flow.
- ***Live keyboard:*** You can check or change the values of variables while a program is running or paused.
- ***PAUSE:*** Put PAUSE statements in your programs for debugging, then remove them later.
- ***PRINT:*** Like PAUSE, you can insert PRINT statements to find out the value of a variable or register at any point in the program.
- ***TRACE ALL:*** This makes the computer "trace" program flow and variable assignments on the current PRINTALL IS device – CRT or external printer.
- ***TRACE PAUSE:*** Causes program execution to pause before executing the specified line. So if you suspect a problem in line 4010, type TRACE PAUSE 4010 before you press [RUN].
- ***WAIT:*** Some instrument lockups are caused by a computer that's too fast for the instrument, or vice versa. While not the most elegant solution, plugging WAIT statements into your program to allow "settling" time just *may* get you running.

Hints for Debugging

Be logical when you're ferreting out bugs. Break the code into small segments and check each segment individually, working from the beginning of the program to the end.

Some errors such as mismatched structures occur at pre-run; others at runtime. Here are some hints for debugging:

- ***Fix bugs as you find them:*** Fix each bug as soon as you find it, so it doesn't cause other problems.
- ***Use test cases:*** Use values that test inside and outside all the limits you *think* you have in the program.
- ***Determine if a bug is repeatable:*** Is it intermittent or does it occur all the time, with every test case?

- *Look for simple mistakes:* Somehow, the simpler they are, the easier they are to overlook.
- *Don't assume anything:* Verify the value of variables before and after subroutine calls. Step through the program and make sure everything works the way you designed it.
- *Go home:* Sometimes all it takes is a good night's sleep and a fresh outlook to find a problem.

Document the Program

Once the program is written, it's easy to hit the [RUN] button and let it work while you take a well-earned break. Or to forget it entirely as you move on to new challenges.

Not so fast!

You still have to *document* the program, to make it easy for others (and yes, you too) to use.

Two things you want right up front are the program's name and the date it was written – or revised.

```
10 ! RE-STORE "FILTER_TST"  
20 !  
30 !LAST REVISED: July 4, 1988
```

Then you'll want a general description of what the program does. You needn't write *A Tale of Two Cities* – just an overall statement will be fine.

You'll probably want a similar description for each subprogram, since these can be used in more than one program.

If you've chosen subroutine labels that make sense, you may not need to add remarks. Then again, for the sake of clarity, a brief comment can't hurt, can it?

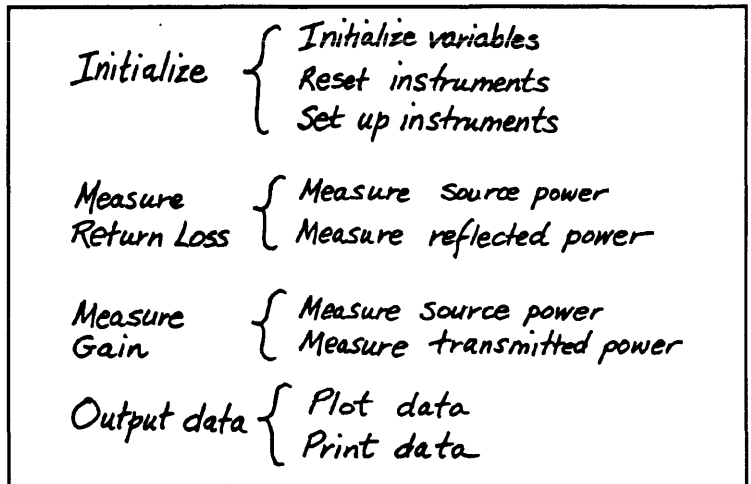
Review Quiz

1. Put these steps in the *proper* order:
Write code.
Create Warnier-Orr diagram.
Document the program.
Test and debug.
Determine a manual procedure.
Check out the hardware setup.
Determine subroutines and subprograms.
2. How would you test this section of code? What values for Stat would you use?

```
50!  
60 Stat = SPOLL (@Mna)  
70 IF NOT BIT (Stat,5) THEN GOTO 60  
80!
```

3. What's one way to eliminate instrument lockups resulting from timing problems (a computer that's too fast for an instrument, or vice versa)?

4. The following Warnier-Orr diagram is for testing a microwave amplifier. Write the main program and the subroutine calls.



Don't worry about filling in the actual program steps in this example; just put a remark (!) line where the code would be, like this:

```
100 Init:!  
120 !  
130 RETURN
```

Laboratory Exercise

Poor Fitz Binkle! He's written a program that purports to get data from the data file called "8590_TRACE" on the disk of examples. He wants to print and plot the data (with a meaningful plot), then store the data in an ASCII file under another name.

Unfortunately for Binkle, though, his program is not very structured, and is riddled with errors. Can you structure the program and debug it for him? Binkle is at his wit's end, and would be ever so grateful. (The program is on your disk of examples as "FITZ_FOLLY".)

You'll need all the debugging tricks at your command to help out Binkle. When you're done, compare your solution with SOL_LAB20 on your examples disk.

Index

Volume 1 contains Introduction and Lessons 1-20.
Volume 2 contains Lessons 21-30 and Appendixes.

A

- A image specifier, 9-11, 25-26, 26-21
- ABORT, 13-4
- ABORT, bus sequence, 21-18
- ABS, 1-19
- Absolute value, 1-19
- ACS, 1-19
- Addition, 1-15
- Address, changing, 12-12
- Address, device, 12-10, 13-1
- Address, instrument, 12-8
- Address of file, 10-7
- Address, secondary, 12-12
- Address, setting, 12-8
- Address, using with ENTER, 15-3
- Addressing multiple instruments, 22-1
- Addressing non-active controller, 22-17
- ALLOCATE, 5-6, 8-8
- Alpha display, 19-2
- ALPHA ON/OFF, 19-11
- Alpha plane, 19-11
- [Alt] key, 1-5
- AND, 6-14
- Answers to review questions, A-1
- Arc sine, 1-19
- Arccosine, 1-19
- Arctangent, 1-19
- Argument, 1-19
- Array separators, 25-4
- Arrays, 8-1
- Arrow keys, 1-9, 1-12
- ASCII characters, 5-20, 21-20
- ASCII data format, 26-35
- ASCII file, 3-25, 10-4, 10-7
- ASCII file, data size in, 10-9
- ASCII value, returning, 5-20
- ASCII vs. BDAT data files, 17-12
- ASN, 1-19
- ASSIGN, 10-13
- ASSIGN, specifying data size with, 26-15
- ASSIGN with attributes, 25-8
- ASSIGNed attributes and ENTER, 26-14
- Assigning data to variables, 5-8
- Assigning numbers to variables, 4-6
- Assigning strings to variables, 5-4
- Asterisk as image specifier, 25-23, 26-19
- Asterisk, multiplication with, 1-16
- Asterisk, using to close I/O path, 10-15
- Asterisk, using with array, 8-11, 8-20
- At sign (@) as image specifier, 25-30, 26-21
- ATN, 1-19
- ATN line, 11-14
- Attention, 11-14
- Attributes, assigning, 25-12
- Auto-starting a program, 3-24

AXES, 19-22
Axes lines, graphics, 19-22

B

B image specifier, 9-13, 25-27, 26-24
[BACK SPACE] key, 1-12
BASE, 8-19
Base element of array, 8-4
BASIC history, 2-6
BCD interface, 11-4
BDAT file, 10-7
BDAT file, data size in, 10-8
BDAT files and storage space, 29-2
BDAT vs. ASCII data files, 17-12
BEEP, 1-24
BENCHMARK example program, 29-6
Benchmarking, 29-3
Binary data file, 10-7
Binary image with ENTER, 26-24
Binary image with OUTPUT, 25-27
Binary programs, Intro-4
Binary programs, listing, 3-11
BIT, 16-12
Bit-parallel, byte-serial transfer, 11-6
Bits, 11-5
Blank COM, 24-13
Branch, 6-1
BUBBLE example program, 8-15
BUFF_PTRS example program, 27-12
BUFFER, 27-9
Buffer pointers, 27-11
Buffers, 27-1
Bus control and data line register, 16-7
Bus control level, 21-3
Bus lines, HP-IB, 11-12
Bus management lines, HP-IB, 11-14
Bus messages, HP-IB, 21-1, 21-4, D-1

Bus messages, sending, 21-19
Byte, 11-5, 25-18
BYTE attribute, 26-15, 26-24
Byte, entering, 26-24
Byte/Rec, 10-7
Byte, specifying with image, 25-27
Byte, status, 16-20

C

Cable length, HP-IB, 12-18
Cables, European, 12-15
Cables, HP-IB, 12-15
Calculator mode, 2-2
CALL, 6-8, 24-2
CALL and interrupt, 28-2
Capability codes, HP-IB, 12-7, B-1
[CAPS LOCK] key, 1-4
[CAPS LOCK] key, simulating, 30-2
CASE, 6-27
CAT, 3-15, 3-26
CAT TO, 3-27
CAT to determine subprograms, 30-5
Catalog display, 3-15
Catalog of programs, 3-26
Centronics interface, 11-4
Changing address, 12-12
Changing variable while running, 7-11
Character size, graphics, 19-33
CHR\$, 5-21, 5-23
CLEAR, 13-8
CLEAR, bus sequence, 21-17
[CLEAR LINE] key, 1-13
CLEAR SCREEN, 1-13, 2-21
[CLEAR SCREEN] key, simulating, 30-1
Clearing graphics, 19-5
Clearing screen, alternate method, 18-20
CLIP OFF, 19-31

CLIP ON, 19-31
 Clipping, 19-30
 Closing path to file, 10-15
 [CLR I/O] key, 28-2
 [CLR SCR] key, 1-13
 CMD, 21-19
 CMD bus message, 21-9
 Co-processor, 29-11
 COM, 5-6, 8-8, 24-13
 COM area, used with ASSIGN, 25-10
 COM blocks and subprograms, 30-7
 COM, increasing speed with, 29-11
 Comma as array separator, 25-7
 Comma as separator, 25-3
 Comma, in printing, 3-10
 Comma, overridden by PRINT USING, 9-9
 Comma, spacing with, 5-17, 9-2
 Comma, using with ENTER, 26-2
 Command, 2-10
 Command, instrument, 14-5, 14-7
 Command vs. data, separating, 11-14
 Common block, 24-13
 COMMONER example program, 24-14
 Comparisons, 6-11
 Compatibility, HP-IB, 11-5
 COMPLEX, 8-8
 Complex number, 4-4
 Concatenating strings, 5-10
 Connecting HP-IB devices together, 12-16
 Connector, HP-IB, 12-2
 CONT with TRANSFER, 27-5
 Context, 24-2, 24-15
 [CONTINUE] key, 3-3
 CONTROL, 23-5
 [CONTROL] key, 1-5
 Control registers, HP-IB, 16-3, 23-5, C-1
 CONTROL, re-positioning pointers, 27-15
 Control table, buffer, 27-11
 Controller, 11-9
 Controllers, multiple, 22-9
 Converting ASCII value to real number, 5-24
 Converting characters to numbers, 5-20
 Converting lowercase and uppercase, 5-22
 Converting number to string, 5-24
 Converting numbers to characters, 5-21
 Converting string to numbers, 5-19
 Copying program lines, 2-18
 COPYLINES, 2-18
 COS, 1-19
 Cosine, 1-19
 COUNT with TRANSFER, 27-5, 27-21
 CREATE, 10-5, 10-10
 CSIZE, 19-33
 [CTRL] key, 1-5, 18-6
 Custom bus messages, 21-1
 Cutting execution time for I/O, 14-6

D

D image specifier, 9-6, 25-23, 26-19
 DAB mnemonic, 21-14
 DATA, 4-6
 Data and commands, separating, 11-14
 DATA bus message, 21-9
 Data byte, 21-14
 Data files, 10-5, 17-3
 Data formats, 17-2
 Data formatting, 25-1
 Data, instrument, 17-2
 Data lines, HP-IB, 11-12
 Data pointer, 4-9
 Data rate, HP-IB, 11-7
 Data, storing on mass storage, 10-2, 17-11
 Data transfer interrupt, 28-9
 Data transfer speed, HP-IB, 11-11
 Data valid, 11-13
 Data vs. program, 10-5

- Datacomm interface, 11-4
- DATE\$, 1-24
- Date, returning, 1-24
- DAV line, 11-13
- DCL mnemonic, 21-14
- DEALLOCATE, 8-8
- Debugging, 20-19
- Decisions, making, 6-10
- Declaring a variable, 4-13
- DEF FN, 7-14
- DEG, 1-19
- Degrees mode, 1-19
- [DEL CHR] key, 1-12
- DELAY, 25-17
- Deleting a subroutine, 24-25
- Deleting characters, 1-12
- DELIM with TRANSFER, 27-5
- DELSUB, 24-25, 30-8
- Device address, 12-10
- Device clear, 21-14
- DIGITIZE, 28-12
- DIM, 5-6, 8-5
- Dimensions of array, 8-2, 8-5
- Dimensions of array, finding, 8-18
- DIO lines, 11-12
- DIO lines and parallel poll, 22-6
- Directing output to device, 3-27
- Disabling front-panel control, 13-5
- Disk, 3-19, 10-1
- Disk drive, addressing, 3-17
- Disk, using, Intro-5
- DISP, 2-13, 2-22, 3-9
- DISP USING, 9-16
- Display, 1-8
- Display line, 1-10, 3-9
- Distance between devices on HP-IB, 11-7
- DIV, 1-20
- Division, 1-16

- DOS files with HP-UX files, 10-12
- DRAW, 19-6
- Drawing a line, 19-6, 19-8
- DROUND, 1-20
- DUMP DEVICE IS, 19-42
- DUMP GRAPHICS, 19-42
- DVAL, 5-23
- DVAL\$, 5-23
- Dvorak keyboard, 1-7

E

- E image specifier, 9-6, 25-24
- E in number display, 1-17
- e, base, 1-20
- EDIT, 13-11
- EDIT KEY, 18-4, 18-13
- Edit mode, 2-3, 2-15
- Edit mode, leaving, 3-10
- Editing a line, 2-16
- Editing softkeys on PC, 18-13
- ELSE, 6-15
- Empty pointer, buffer, 27-11
- ENABLE INTR, 16-17, 16-28, 28-16
- Enabling interrupt, 16-17, 16-28, 28-16, 28-20
- END, 2-14
- END and HP-IB, 26-3
- END as termination for ENTER, 26-26
- END IF, 6-15
- END LOOP, 7-9
- End or identify, 21-14
- End or identify line, 11-15
- END SELECT, 6-29
- END termination, 26-11
- END WHILE, 7-8
- END with files, 25-8
- END with free-field OUTPUT, 25-7
- END with HP-IB, 25-8, 25-17, 25-37

END with images, 25-36
 END with TRANSFER, 27-5
 End-of-file pointer, 25-8, 26-14
 End-of-record with TRANSFER, 27-5
 ENTER, action of, 15-3
 ENTER, bus sequence, 21-17
 ENTER data from disk, 10-16
 ENTER format, 26-1
 [ENTER] key, 1-2
 ENTER, simulating, 21-21
 ENTER, terminating, 26-18
 ENTER USING, 9-16, 26-16
 ENTER, using with device address, 15-3
 ENTER, using with device name, 15-3
 ENTER, using with I/O path, 15-4
 ENTER, using with OUTPUT, 15-8
 ENTER with ASSIGNED attributes, 26-14
 ENTER with data, 17-4
 ENTER with image, 26-16
 ENTER with instrument, 15-2
 Entering a program, 13-11
 Entering new lines, 2-17
 EOF pointer, 25-8
 EOI, 26-3, 26-9
 EOI, adding with ASSIGN, 25-17
 EOI line, 11-15
 EOI mnemonic, 21-14
 EOI, redefining, 26-26
 EOI, terminating ENTER, 26-18
 EOI termination, 26-11
 EOL OFF, 25-18
 EOL sequence, changing with ASSIGN, 25-15
 EOL sequence, suppressing, 25-4, 25-7, 25-36
 EOR with TRANSFER, 27-5
 Erasing, 1-12
 Erasing a program from memory, 3-23
 Erasing program from mass storage, 3-27
 Erasing softkey definitions, 18-12
 ERRDS, 28-10
 ERRL, 28-10
 ERRM\$, 28-10
 ERRN, 28-10
 Error, ignoring, 30-2
 Error message, display, 1-11
 Error trapping, 28-9
 ESZ image specifier, 26-20
 ESZZ image specifier, 9-7, 25-24, 26-20
 ESZZZ image specifier, 25-24
 Examples disk, Intro-1
 Exclamation point (remark), 3-5
 EXIT IF, 7-9
 EXP, 1-20
 Exponent of number, 1-17
 Extended addressing, 22-3
 Extended register, 16-22
 Extending distance of HP-IB, 30-3

F

Fields, output, 9-2
 File, 10-6
 File, ASCII, 3-25, 10-4
 File, creating, 10-10
 File, data, 10-5, 17-3
 File name, 3-23, 10-7
 File pointer, 25-8
 File size requirements, 10-8
 File type, 10-7
 Files and I/O paths, 26-14
 Fill pointer, buffer, 27-11
 FILTER_TST example program, 20-11
 FITZ_FOLLY example program, 20-24
 Fixed-field data format, 17-2
 Fixed-field OUTPUT, 25-19

- Floating-point data format, 26-38
- Floating-point math board, 29-11
- Flowcharts, 6-22
- FN, 7-14, 24-2
- FNEND, 7-14
- FOR-NEXT, 3-8, 7-1
- Form feed, 3-13
- Formal parameter list, 24-9
- Format of data, 17-2
- FORMAT OFF attribute, 25-12, 26-38
- FORMAT ON attribute, 25-12, 26-15
- Format, disk, 3-19
- Format, using instrument, 26-35
- FORMAT_TST example program, 25-13
- Formatted ENTER, 26-1
- Formatted OUTPUT, 25-1
- Formatting, 9-1
- FRACT, 1-20
- Fractional portion of number, 1-20
- FRAME, 19-14
- Free-field data format, 17-2
- Free-field ENTER, 26-2
- FRIENDS example program, 8-12
- Function, 2-10, 6-8, 7-13
- Function, calling, 24-2

G

- GCLEAR, 19-5
- GDU's, 19-3
- GET, 3-25
- GET mnemonic, 21-14
- Getting a program, 3-25
- GINIT, 19-5
- Go to local, 21-14
- GOSUB, 6-4
- GOSUB and interrupt, 28-2
- GOTO, 6-1, 6-4

- GPIB, 11-4, 11-7
- GPIO, 11-4, 11-7
- Graphic display units, 19-3
- Graphics, 19-1
- Graphics application, 19-12
- GRAPHICS OFF, 19-11
- GRAPHICS ON, 19-2, 19-11
- Graphics plane, 19-2
- Graphics scale, self-computing, 19-40
- Graphics, turning off, 19-11
- GRID, 19-26
- Grid, placing in graphics, 19-26
- Group execute trigger, 21-14
- GTL mnemonic, 21-14
- GUESS_GAME example program, 7-5

H

- H image specifier, 25-25, 26-20
- Handshake, HP-IB, 21-25
- Handshake lines, 11-13
- Hard clip area, 19-3, 19-31
- Hardware, increasing speed with, 29-11
- Hardware, installing, 12-1
- Hardware priority, 28-27
- Hewlett-Packard Interface Bus, 11-5
- History of HP-IB, 11-2
- HP 438A Dual Sensor Power Meter, 15-6
- HP 3325B Synthesizer, 20-3
- HP 3326A Two-Channel Synthesizer, 13-9
- HP 3456A Digital Multimeter, 20-4
- HP 3457A Multimeter, 27-18
- HP 37201A HP-IB Extender, 30-4
- HP 37204A HP-IB Extender, 30-3
- HP 3852A Data Acquisition and Control System, 28-13
- HP 6030A Autoranging Power Supply, 24-20

HP 6624A Power Supply, 14-3
 HP 8340B Synthesized Sweeper, 21-2
 HP 8350B Sweep Oscillator, 30-12
 HP 8510B Network Analyzer, 26-32
 HP 8590A Portable Spectrum Analyzer , 17-7
 HP 8720A Microwave Network Analyzer, 22-4
 HP 8753 Network Analyzer, 16-8
 HP 8757A Scalar Network Analyzer, 30-10
 HP 8980A Vector Analyzer, 18-14
 HP 98365A, 29-11
 HP-IB addresses and switch settings, 12-13
 HP-IB bus messages, D-1
 HP-IB capabilities, 12-4
 HP-IB interrupts, external, 28-16
 HP-IB lines, 21-16
 HPIB_LINES example program, 16-13
 HP-IB status and control registers, C-1
 HP-UX and DOS files, 10-12
 HP-UX file, 10-7
 HP-UX file, data size in, 10-9

I

I/O path, 10-13
 I/O path, closing, 10-15
 I/O path, finding information about, 25-11
 I/O path name, 10-13
 I/O path name for instrument, 13-3, 14-6
 I/O path, using with ENTER, 15-4
 Identifying HP-IB devices, 12-1
 IEC-625, 11-7
 IEC-625 cables, 12-15
 IEEE-488, 11-4, 11-7
 IEEE-488 interface capability codes, B-1
 IEEE-488 mnemonics, 21-13, D-7
 IF-THEN, 6-10, 6-16
 IF-THEN with AND-OR, 6-14
 IF-THEN with END IF, 6-15
 IF-THEN with END IF and ELSE, 6-15
 IFC line, 11-14
 IFC mnemonic, 21-14
 IMAGE, 9-15, 17-18, 25-20
 Image specifiers, 9-6
 Image specifiers, boundaries for, 26-29
 Image specifiers for PRINT USING, 9-13
 Image specifiers for strings, 9-11
 Image, evaluating, 25-21
 Image, using, 9-15
 IMAGE with ENTER, 26-16
 Image with OUTPUT, 9-6, 17-17, 25-23
 Inbound transfer, 27-6
 INDENT, 7-7
 INITIALIZE, 3-17, 10-4
 Initializing a disk, 3-17, 10-3
 Initializing a disk on a PC, 3-21
 Initializing graphics, 19-5
 INPUT, 2-11, 4-10, 4-13, 5-8, 8-9
 Installing HP-IB hardware, 12-1
 Instrument command, 14-7
 Instrument command, specifying, 14-9
 Instrument data, 17-2
 Instrument data format, 26-35
 Instrument features, using, 15-14
 INT, 1-20
 INTEGER, 4-4, 4-13, 4-15, 8-7
 Integer number, 4-4
 Integer portion of number, 1-20
 Integer, returning after division, 1-20
 Integer, using for speed, 29-8
 Integer value, returning, 5-23
 Interface, 11-3
 Interface clear, 11-14, 21-14
 Interface select code, 12-9
 Interleave factor, 3-19
 Internal data format, instrument, 26-42
 Interrupt, 16-15, 28-1

Interrupt, disabling, 28-18
Interrupt enable mask, 16-17, 16-28,
22-13, 28-17
Interrupt enable register, 22-20
Interrupt, non-active controller, 22-12
Interrupt status register, 16-6, 16-17
Interrupt, using softkey for, 18-17
Item separator, 26-2
Item terminator, 26-2
IVAL, 5-23
IVAL\$, 5-23

K

K image specifier, 9-7, 9-11, 25-24, 25-26,
26-19, 26-21
KEY LABELS ON/OFF, 1-14, 16-4, 23-5
Key, simulating, 30-1
Keyboard, 1-2, 1-6
Keyboard area, display, 1-10
Keyboard, live, 7-11, 20-20
Keyboard overlay, Intro-4, 1-5
Keypad, 1-4
Keys, system, 18-11
Keyword, 2-4
KNOBX, 28-11
KNOBY, 28-11

L

L image specifier, 9-13, 25-32, 26-21
Label, 6-3, 19-30
Label, adding to disk, 10-4
Label, softkey, 18-18
LABEL USING, 9-16
Labelled COM, 24-13
Labelling graphics, 19-29, 19-32
LAG mnemonic, 21-15

LDIR, 19-34
Learn string, instrument, 26-46
LEN, 5-16, 5-23
Length of string, 5-16
LET, 2-12, 4-6, 5-4, 8-9
Levels of control, 21-3
LGT, 1-20
Line label, 6-3
Line numbers, 2-4
LINE TYPE, 19-10
Line type, choosing, 19-10
Linear pattern, 12-17
LINPUT, 5-8
LIST, 3-10, 3-27
LIST BIN, 3-11
LIST KEY, 18-3
Listen address, 21-5, 21-15
LISTEN bus message, 21-10
Listener, 11-9
Listing a binary program, 3-11
Listing a program, 3-10
Listing softkeys, 18-3
Literal as image specifier, 26-23
Literal in string image, 25-26
Live keyboard, 7-11, 20-20
LLO mnemonic, 21-15
LOAD, 3-24
LOAD KEY, 18-12
Loading a program, 3-24
LOADSUB, 30-5
LOADSUB ALL, 30-5
LOADSUB FROM, 30-6
LOCAL, 13-7
LOCAL, bus sequence, 21-17
[LOCAL] key, 13-6
LOCAL LOCKOUT, 13-6, 21-4, 21-15
LOCAL LOCKOUT, bus sequence, 21-18

Local mode, 13-6, 14-2
Local variables, 7-15
Local variables and COM, 24-18
LOG, 1-20
Logarithm of number, 1-20
Logging an event, 28-21
Look-up table, 29-7
Loop, 6-2, 7-1
Loop counter, 7-2
LOOP-END, 7-9
Loop, filling array with, 8-9
Loops, 3-8
LORG, 19-32
Lowercase, converting to, 5-22
LWC\$, 5-22

M

M image specifier, 9-6, 25-23, 26-19
MAGIC example program, 8-18
Magic square, 8-22
Masking, 16-27, 16-29
Masking the SRQ, 16-20
Mass storage, 3-15
Mass storage, addressing, 3-17
MASS STORAGE IS, 3-15, 10-3
Mass storage, specifying, Intro-5
Mass storage, using, 10-1
Math functions, 1-14
Mathematics, order of expressions, 1-22
MAX, 1-20
Maximum value, returning, 1-20
MAXREAL, 1-20
MC 68881 co-processor, 29-11
Memory, 3-14
Memory, reserving space in, 8-6
Message and results line, display, 1-11
Message, displaying, 2-22

MIN, 1-20
Minimum value, returning, 1-20
MINREAL, 1-20
Minus sign (–), 26-28
MLA bus message, 21-11
MLA mnemonic, 21-15
Mnemonics, IEEE-488, D-7
MOD, 1-20
MORNING example program, 6-25
MORTGAGE example program, 4-12
Mouse, 28-11
MOVE, 19-6
MOVELINES, 2-18
Moving program lines, 2-18
MSI, Intro-5, 3-15, 10-3
MSI_KEYS example program, 18-8
MTA bus message, 21-10
MTA mnemonic, 21-15
Multiple images, 9-9
Multiple instruments and controllers, 22-1
Multiple listeners, 22-2
Multiplication, 1-16
My listen address, 21-11, 21-15
My talk address, 21-10, 21-15

N

Name, using with ENTER, 15-3
Named buffer, 27-8
Named COM, 24-13
NDAC line, 11-13
Negative step in loop, 7-3
Nested images, 25-36
New lines, entering, 2-17
NEXT, 7-1
Non-active controller, 22-12
Not data accepted, 11-13
Not ready for data, 11-13

NPAR, 24-10
NPAR_COUNT example program, 24-10
NRFD line, 11-13
Null character, 26-15
Null string, 5-9
NUM, 5-20, 5-23
Number builder, 17-5, 26-3
Number, building from ASCII stream, 26-3
Number, converting to ASCII, 5-23
Number of devices on HP-IB, 11-7
Numeric image for OUTPUT, 25-23
Numeric image with ENTER, 26-19
Numeric variable and number builder, 17-6

O

ON, 6-24, 6-27, 28-3
ON CDIAL, 28-4
ON CYCLE, 28-4, 28-8
ON DELAY, 28-4, 28-8
ON END, 28-4, 28-9
ON EOR, 28-5, 28-9
ON EOT, 28-5, 28-9
ON ERROR, 28-5, 28-9
ON HIL EXT, 28-5
ON INTR, 16-17, 28-6, 28-16
ON KBD, 18-21, 28-6
ON KEY, 18-17, 28-6
ON KNOB, 28-6, 28-11
ON SIGNAL, 28-6
ON TIME, 28-7
ON TIMEOUT, 28-7
Opening path to file, 10-13
OPTION BASE, 8-4
Option base, determining, 8-19
OPTIONAL, 24-10
OR, 6-14
Order of evaluation, 1-22

Outbound transfer, 27-7
Output area, 3-9
Output area, display, 1-8
OUTPUT, bus messages for, 21-11
OUTPUT, bus sequence, 21-17
OUTPUT, formatting, 25-1
OUTPUT, free-field, 25-2
OUTPUT of data, 14-7, 17-3
OUTPUT, rounding, 25-2
OUTPUT, simulating, 21-20
OUTPUT to I/O path, 10-14, 14-6
OUTPUT to instrument, 14-5
OUTPUT to keyboard, 30-1
OUTPUT to named device, 14-6
OUTPUT, used with ENTER, 15-8
OUTPUT USING, 9-16, 17-17, 25-19
Overscore, meaning, 21-16

P

PANAMA example program, 5-11
Parallel poll, 21-15, 22-5
Parallel poll configure, 21-15
Parallel poll disable, 21-15
Parallel poll enable, 21-15
Parallel poll response byte, 22-8
Parallel poll response mask, 22-20
Parallel poll unconfigure, 21-15
PARAM_PASS example program, 24-5
Parameter list, 24-9
Parameters, finding how many, 24-10
Parameters, optional, 24-10
Parentheses, using in mathematics, 1-21, 1-23
Part of string, 5-12
Partial listing, 3-11
PASS CONTROL, 22-10
PASS CONTROL, bus sequence, 21-18
Pass parameter list, 24-9

Passing by reference, 24-4
 Passing by value, 24-4
 Passing parameters, 24-3
 Passthrough mode, 30-13
 Path name, 10-13
 Path name, using for instrument, 13-3
 PAUSE, 20-20
 [PAUSE] key, 3-3
 PEN, 19-6
 Pen control digit, 19-9
 Pen, graphics, 19-6
 Percent sign (%) as image specifier, 25-32, 26-29
 Period (.) as image specifier, 25-24, 25-32, 26-19
 Personal computer, using, Intro-7
 PI, 1-20
 Pi, value of, 1-20
 Pickwick club, 11-8
 PLOT, 19-8
 PLOTTER IS, 19-13
 Plotter, specifying, 19-13
 Plotting area, scaling, 19-18
 Plotting data, 19-36
 Plus sign (+) as image specifier, 25-32, 26-28
 Pointer, buffer, 27-11
 Pointer, data, 4-9
 Pointers, re-positioning with
 CONTROL, 27-16
 POS, 5-16, 5-23
 Position pointer, I/O path, 26-14
 Position within string, 5-16
 Pound sign (#) as image
 specifier, 9-13, 25-32, 26-28
 Power of 10, 1-17
 PPC mnemonic, 21-15
 PPD mnemonic, 21-15
 PPE mnemonic, 21-15
 PPOLL, 22-7
 PPOLL CONFIGURE, 22-6
 PPOLL mnemonic, 21-15
 PPOLL RESPONSE, 22-19
 PPOLL response byte, 22-8
 PPOLL UNCONFIGURE, 22-9
 PPU mnemonic, 21-15
 Pre-run, 4-2, 20-19
 Pre-run and COM, 24-17
 PRINT, 3-9, 20-20
 PRINT LABEL, 10-4
 PRINT USING, 9-4
 PRINTALL, 23-6
 PRINTALL IS, 1-14
 PRINTER IS, 3-6, 3-27
 Printing, 9-1
 Printing graphics, 19-42
 Printing strings, 9-11
 Printing, trouble-killers for, 3-12
 PRIORITIES example program, 28-23
 Priority, hardware, 28-27
 Priority, software, 28-21
 PROG file, 3-22, 3-26
 Program, designing, 20-1
 Program, documenting, 20-21
 Program, storing on mass storage, 10-3
 Program vs. data, 10-5
 Programs, binary, Intro-4
 PROUND, 1-19, 4-14
 [PRT ALL] key, 1-15
 Pseudocode, 20-9
 PURGE, 3-27

Q
 QUADRATIC example program, 6-17
 Quotation marks, inserting, 5-21
 QWERTY keyboard, 1-6

R

R image specifier, 25-24, 26-19

RAD, 1-20

Radians mode, 1-20

Raising to a power, 1-17

RAND_ROOTS example program, 10-20

Random access of data in file, 10-14, 10-20

Random number, 1-21

RANDOMIZE, 1-21, 9-4

RANK, 8-18

RE-SAVE, 3-25, 10-4

RE-STORE, 3-23, 10-4

Re-using images, 25-35

READ, 4-6, 8-9

READ LOCATOR, 28-12

Reading a status register, 16-3, 23-3

Reading an instrument with ENTER, 15-2

Reading status byte, 16-23

READIO, 23-6

REAL, 4-4, 8-7

REAL data format, 26-38

Real number, 4-3

Rec/File, 10-7

Record, 10-6

Record number, 10-22

RECORD with TRANSFER, 27-5

Records, 3-26

RECOVER, 28-2

Register, 16-2

Register contents, controlling, 23-1

Register control level, 21-3

Register, READIO and WRITEIO, 23-6

Register, status, 16-20

Registers, 23-1

Registers, buffer, 27-11

Registers, HP-IB, C-1

REM, 3-5

Remainder, returning, 1-20

Remarks, 3-5, 20-21

Remarks and storage space, 29-3

Remarks with INDENT, 7-7

REMOTE, 13-5

REMOTE, bus sequence, 21-17

Remote enable, 11-14, 21-16

Remote mode, 13-5

REN, 2-19, 6-3

REN line, 11-14

REN mnemonic, 21-16

Renumber, effect of, 6-3

Renumbering program lines, 2-19

Repeat factor in image, 25-34

REPEAT-UNTIL, 7-4

Repeating a string, 5-24

Repeating a task, 7-1

Replacing a string, 5-10

Replacing part of string, 5-15

REQUEST, 22-18

Requesting service, 16-1

Reserving memory for strings, 5-5

RESET, 13-5

[RESET] key, 13-7, 28-2

RESTORE, 3-23, 4-9

Retrieving data from disk, 17-16

RETURN, 6-4

RETURN with ASSIGN, 30-2

RETURN with function, 7-14

REV\$, 5-11, 5-23

Reversing a string, 5-11

Review quizzes, answers to, A-1

Rewriting softkey definition, 18-4

RND, 1-21, 9-4

ROLL_DICE example program, 6-5

Rounding a number, 1-20, 4-14

Rounding to power of 10, 1-20

RPT\$, 5-24

RS-232-C interface, 11-4
Run indicator, 2-9, 3-4
Run mode, 2-7
Running a program, 2-7, 13-12

S

S image specifier, 9-6, 25-23, 26-19
SAVE, 3-25, 10-4
Saving a program, 3-1, 3-25
Saving instrument data, 17-1
SCRATCH, 2-3, 3-23, 13-11
SCRATCH and COM, 24-17
SCRATCH KEY, 18-12
SDC mnemonic, 21-16
SEC bus message, 21-10
Secondary addresses, 12-12, 21-3, 21-10, 22-3
SELECT-CASE, 6-27
Select code, 3-6, 12-8, 13-1
Selected device clear, 21-16
Semicolon as array separator, 25-7
Semicolon as separator, 25-4
Semicolon, in printing, 3-10
Semicolon, spacing with, 2-22, 5-17, 9-2
Semicolon, using to suppress CR/LF, 14-12
Semicolon, using with ENTER, 26-2
SEND, 21-5, 21-17, 21-19, D-1
SEPARATE ALPHA FROM
 GRAPHICS, 19-11
Separating instrument commands, 14-12
Separators, 25-3
Separators with ENTER, 26-2
SER_ROOTS example program, 10-17
Serial access, 10-14
Serial poll, 16-24
Serial poll disable, 21-16
Serial poll enable, 21-16
Serial poll register, 16-22
Service request, 16-1, 16-7
Service request from non-active
 controller, 22-18
Service request line, 11-14
SET KEY, 18-8, 18-13
Setting the HP-IB address, 12-8
SGN, 1-21
[SHIFT] key, 1-4
SHOW, 19-18
Sign, returning, 1-21
SIGNAL, 28-6
Simulating a key press, 30-1
SIN, 1-21
Sine, 1-21
SIZE, 8-19
Size of array, 8-5
Size of array, finding, 8-19
Slash (/) as image specifier, 9-13, 25-30, 26-23
Slash (/), division with, 1-16
Soft clip area, 19-17, 19-30
Softkey, 1-5
Softkey as typing aid, 18-3
Softkey, changing from program, 18-8
Softkey definitions, erasing, 18-12
Softkey definitions, storing, 18-11
Softkey label, 1-11
Softkey, redefining, 18-4
Softkey, using for program interrupt, 18-17
Softkeys, editing on a PC, 18-13
Softkeys, listing, 18-3
Softkeys, loading, 18-12
Softkeys, returning to default, 18-12
Softkeys, using, 18-1
Software priority, 28-21
Sorting, 8-15
Space, reducing, 29-2
Spacing, 5-18
Spacing for DISP and PRINT, 5-17

SPD mnemonic, 21-16
 SPE mnemonic, 21-16
 Speed, increasing, 29-3
 SPOLL, 16-24
 SPOLL and interrupt, 28-18
 SPOLL, bus sequence, 21-18
 SPOLL, simulating, 21-21
 SQR, 1-21
 SQRT, 1-18, 1-21
 Square root, 1-18, 1-21
 SRQ, 16-2, 16-7
 SRQ, bus sequence, 21-18
 SRQ, detecting, 16-12
 SRQ line, 11-14
 SRQ, masking, 16-20
 SRQ, non-active controller, 22-18
 Star pattern, 12-17
 Statement, 2-10
 Statement control level, 21-3
 STATUS, 16-3, 16-12, 16-23, 23-3
 Status byte, 16-20, 16-23
 Status byte, unmasking, 16-22
 Status change, detecting, 16-12
 Status, detecting with instrument
 command, 16-29
 STATUS of ASSIGNED I/O path, 25-11
 Status register, 16-2, 16-20, 23-2
 Status register, reading, 23-3
 Status registers, HP-IB, C-1
 Status word, 16-22
 STEP, 7-2
 [STEP] key, 2-11, 20-20, 27-12
 Stopping activity on HP-IB, 13-4
 STORE, 3-22, 10-4
 STORE KEY, 18-11
 Storing a program, 3-22, 10-4
 Storing data on mass storage, 10-2
 Storing instrument data, 17-11
 Storing softkey definitions, 18-11
 String image, 25-26
 String overflow error, 5-8
 String variable, 4-4, 5-2
 Strings and number builder, 17-5, 26-9
 Strings in arrays, 8-12
 Strings, printing, 9-11
 Stripping blanks from a string, 5-24
 Structured programming, 8-23, 20-1
 SUB, 6-8, 24-2
 SUBEND, 6-8
 SUBPR_DICE example program, 6-8
 Subprogram, 6-7, 6-9, 7-16
 Subprogram, calling from keyboard, 24-3
 Subprogram, deleting, 30-8
 Subprogram, effect on speed, 29-9
 Subprogram libraries, 30-5
 Subprograms, 24-1
 Subprograms, loading, 30-5
 Subroutine, 6-4, 6-7
 Subroutine, deleting, 24-25
 Subscripted variable, 8-3
 Substring, 5-12
 Subtraction, 1-15
 SUM, 8-21
 Summary bit, 16-22
 Summing elements of array, 8-21
 Switch settings and HP-IB addresses, 12-13
 Switches, address, 12-10
 System controller, 11-10
 System keys, 18-11
 SYSTEM\$, Intro-7
 SYSTEM_EX example program, Intro-5

T

TAD mnemonic, 21-16
Take control, 21-16, 22-10
Talk address, 21-16
TALK bus message, 21-10
Talker, 11-9
TAN, 1-21
Tangent, 1-21
TCT bus message, 22-10
TCT mnemonic, 21-16
Terminating ENTER statement, 26-11
Terminating ENTER with image, 26-26
Termination, changing with ENTER
 USING, 26-28
Termination image specifiers, 25-32
Terminator, number, 26-8
Terminators, 25-3
Terminators with ENTER, 26-2
Tic marks, axes, 19-24
Tic marks, grid, 19-27
Tic marks, major, 19-25
Time, returning, 1-23
TIME\$, 1-23, 18-5
TIMEDATE, 1-23, 18-5
Timeout interrupt, 28-8
TRACE ALL, 20-20
TRACE PAUSE, 20-20
TRANSFER, 27-1
TRANSFER, interrupting, 28-9
TRANSFER parameters, 27-4
TRIGGER, 15-11
TRIGGER, bus sequence, 21-17
Triggering an instrument, 15-10
Trigolator, 19-13
TRIGOLATOR example program, 19-37
TRIM\$, 5-24
Trouble killers, Intro-8

U

UDU's, 19-18
Unconditional branch, 6-2
Unified I/O, 14-6
UNL mnemonic, 21-16
Unlisten, 21-5, 21-10, 21-16
Unmasking status byte, 16-22
Unnamed buffer, 27-9
Unnamed COM, 24-13
UNT bus message, 21-10
UNT mnemonic, 21-16
Untalk, 21-10, 21-16
UNTIL, 7-4
UPC\$, 5-22, 5-24
Uppercase, converting to, 5-22
User defined units, graphics, 19-18

V

VAL, 5-19, 5-24
VAL\$, 5-24, 17-4
Variable, 2-11, 4-2
Variable, array, 8-11
Variable name, 4-3
Variable name, using for instrument, 13-2
Variable, rules for, 4-5
Variable, specifying for ENTER, 15-5
Variable, subscripted, 8-3
Variable type, 4-3
Variable within function, 7-15
VIEWPORT, 19-14
Viewport, setting, 19-14

W

W image specifier, 26-24
WAIT, 2-21, 20-20
WAIT with TRANSFER, 27-5, 27-15
Warnier-Orr diagram, 20-6
WHILE-END, 7-8
WINDOW, 19-18
WORD, 25-18
WORD attribute, 26-3, 26-15, 26-24
Word, entering, 26-24
Word, specifying with image, 25-28
WRITEIO, 23-6
Writing a program, 2-3

X

X image specifier, 9-7, 9-11, 25-30, 26-23

Y

Y image specifier, 25-29, 26-25

Z

Z image specifier, 9-6, 25-23, 26-19

Numbers

3457_READ example program, 27-21
3457_TRANS example program, 27-19
8510_ASCII example program, 26-36
8510_DATA example data file, 26-50
8510_INTL example program, 26-42
8510_LEARN example program, 26-46
8510_REAL example program, 26-38
8590_GET example program, 17-17
8590_STORE example program, 17-12
8590_TRACE example data file, 17-19
8720_PASS example program, 22-11
8753_ERRS example program, 16-26
8753_INTR example program, 16-15, 16-22



**HEWLETT
PACKARD**

82302-90001
Printed in U.S.A. 10/88
English