HP BASIC Language Processor

Programmer's Reference Guide

HEWLETT PACKARD

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.

MS is a U.S. trademark of Microsoft Corporation.
SIDEKICK and SUPERKEY are registered trademarks of Borland International, Inc.

© 1987 by Hewlett-Packard Co.

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

Contents

Chapter 3

Programming Techniques

Chapter 4

Graphics Techniques

1 Introduction

Contents

1 Introduction

Welcome to the HP BASIC Language Processor.

This guide will help you learn about setting up and using your HP BASIC Language Processor software. It will guide you through the software installation process and then will help you become familiar with various programming techniques of the HP BASIC Language Processor. Later, you can use it as a reference. If you are already familiar with HP BASIC Series 200/300 workstations, you should read appendix B ("Using the HP BASIC Language Processor in the MS-DOS Environment"), appendix C ("Utilities"), appendix E ("Keyboards"), and appendix F ("The Configuration File") after you complete the software installation in chapter 2.

What Is in This Guide

You will find instructions for getting started, including procedures for:

- Software installation.
- Programming techniques.
- Graphics techniques.
- Interface techniques.
- Using Shared Resource Management (SRM).

Note 👆 You will find the software installation easier to follow if you are familiar with MS-DOS. If you are not familiar with MS-DOS, the procedures recommend a command name that you can look up in your MS-DOS manual when you need to perform MS-DOS functions.

Additional information is provided in appendixes:

- Using the HP BASIC Language Processor in the MS-DOS Environment.
- Utilities.
- List of Binaries.
- List of Keyboard Functions.
- Configuration.
- List of Programs On the Manual Examples Disc.
- Error list.

What Is in This Chapter

This chapter describes:

- What is in the manual.
- Features of the HP BASIC Language Processor.
- Hardware and software requirements.
- What is in the box.
- The HP BASIC Language Processor discs.
- Other manuals that you may find useful.

Features of the HP BASIC Language Processor

The HP BASIC Language Processor enables you to use your personal computer to develop and run sophisticated BASIC language programs. It provides the emulator program that enables the HP 82321A Language Processor Card to be used in your personal computer. The features include:

- **HP Series 200/300 Basic Language Workstation Emulation:** The HP BASIC Language Processor enables your personal computer to look and perform like an HP Series 200/300 BASIC Language Workstation. A medium resolution graphics display can also be emulated.

- **MS-DOS Environment:** The HP BASIC Language Processor is similar to a typical MS-DOS language system. It accommodates the MS-DOS environment while retaining the HP Series 200/300 BASIC interface and features. MS-DOS allows access to IO devices. The differences are explained in detail in appendix B.

- **Customizing Binaries:** Binaries are optional enhancements to the HP BASIC system. They include both language extensions and drivers. By selectively loading binaries after booting, you can obtain the features you need without using excessive memory.

- **Performance:** HP BASIC provides the performance that enables you to run sophisticated programs.

- **Background Operations:** HP BASIC can operate in the background enabling other personal computer applications to run simultaneously in the foreground. Refer to appendix B for more information.

Hardware and Software Requirements

To set up the HP BASIC Language Processor you need an MS-DOS computer that includes:

- A minimum of 256k bytes of RAM.
- A dual flexible disc drive system with at least one high-capacity (1.2M byte or greater) drive, or a hard disc system with at least one flexible disc drive.
- The HP 82321 Language Processor Card.
- The HP BASIC Language Processor software discs.
- A Monochrome Plus, Multimode, or Enhanced Graphics Display (EGA) adapter or equivalent, plus the appropriate monitor.

Note

EGA emulation of the 9836C color display requires at least 512K bytes of PC RAM.

What Is in the Box

The HP BASIC Language Processor package includes:

- This manual: *HP BASIC Programmers Reference Guide*.
- *BASIC 5.0 Language Reference A–L* (Volume 1).
- *BASIC 5.0 Language Reference M–Z* (Volume 2).
- *Key Function and Switch Configuration Guide*.
- *Basic 5.0 Condensed Reference*.
- Four keyboard overlays.
- Four 5.25-inch HP BASIC Language Processor software discs.
- One 3.5-inch HP BASIC Language Processor software disc.

Labels and Descriptions of 5.25-Inch HP BASIC Discs

Label	Description
Disc one: HP BASIC emulator and PC utilities.	The emulator program plus file and configuration utilities.
Disc two: HP BASIC system disc.	HP BASIC operating system.
Disc three: Binaries and drivers.	Binaries for additional features (drivers for devices).
Disc four: Manual examples and selected HP BASIC Utilities.	Examples from the manual.

The single 3.5-inch disc contains all of the above.

Other Manuals You May Find Useful

If you would like additional detail or information, you may want to use these supplemental manuals available from Hewlett-Packard.

Manual Name	Part Number	Description
Basic 5.0 Programming Techniques	98613-90012	Detailed description of programming.
Basic 5.0 Graphics Techniques	98613-90032	Detailed description of graphics.
Basic 5.0 Interface Techniques	98613-90022	Detailed description on interfaces.
Installing, Using and Maintaining the BASIC 5.0 System	98613-90092	Information on installing, using, and maintaining BASIC on a series 200/300 system.

Manual Name	Part Number	Description
HP-IB Interface Manual	82990-90001	Detailed description of HP-IB interfaces.
SRM System Manager's Guide	98619-90032	General system maintenance.
SRM Hardware Installation Manual	98619-90021	Hardware installation.
SRM Software Installation Manual	98619-90071	Installing the SRM operating system.

Information on recent changes to HP BASIC can be found in a file called README on disc one. You should read this file before you install HP BASIC. Use the MS-DOS TYPE or PRINT command to list the file.

2 Software Installation

Contents

2   Software Installation

This chapter covers:

- Installing the HP BASIC Language Processor on your hard disc or high-capacity (1.2 Mbyte or greater) floppy disc. The install process consists of the following:

  - Installing the HP BASIC Language Processor.
  - Booting the HP BASIC Language Processor.
  - Customizing your HP BASIC system by adding binaries.
  - Saving the system you create.

- Testing your system.

In order to install and use the HP BASIC Language Processor, your PC system must have one of the following combinations of disc drives:

- Dual floppy disc drives, at least one of which must be a high-capacity drive.
- At least one floppy disc drive and a hard disc drive.

If you do not have any of these combinations, you will not be able to install the HP BASIC Language Processor.

The Configuration File

Your HP BASIC system builds a standard configuration file during the installation process. The following table shows what the standard configuration file will provide for your HP Basic Language Processor system.

HP BASIC Select Code	Device	Configuration Notes
1	CRT (Alpha)	
2	Keyboard	Includes HP-HIL mouse and knob.
3	CRT (Graphics)	Monochrome Plus Adapter 400h × 300v B/W.
		Multimode Adapter 512h × 390v B/W.
		Enhanced Graphics Adapter 512h × 350v Color.
7	Built-in HP-IB	HP BASIC interrupt level = 3.
9	COM1 (RS-232) (Optional)[†]	Baud = 9600
		Parity = None
		Char Length = 8
		Stop bits = 2
		HP BASIC interrupt level = 3
		Modem Status Lines CD, RI, DSR, and CTS disabled.
15	Internal Discs	Behave as HP-IB disc drives at primary address 00 to HP BASIC. Refer to the next table for a correlation between HP BASIC MSUS and MS-DOS drive identification.

[*] Must match the settings on the HP 82990 HP-IB card.

[†] Optional interfaces are not configured if they are not connected to your system.

HP BASIC Select Code	Device	Configuration Notes
19	MS-DOS Port	Behaves as a GPIO device to HP BASIC.
23	COM2 (RS-232) (optional)[†]	Same configuration as COM1.
24	HP-IB PC Card 1 (optional)[†]	PC Select Code = 7*
		System Controller = Yes*
		HP BASIC Interrupt Level = Disabled
		PC Interrupts = Disabled
25	HP-IB PC card 2 (optional)[†]	Not configured. Refer to appendix F for more information.
26	LPT1 (printer) (optional)[†]	Behaves as a GPIO device to HP BASIC.

* Must match the settings on the HP 82990 HP-IB card.

[†] Optional interfaces are not configured if they are not connected to your system.

Any GPIO or SRM cards will also be configured to the select codes and options set by switches on those cards.

The following table shows the correlation between HP BASIC Mass Storage Unit Specifier (MSUS) and MS-DOS (Drive ID) for the default configuration for internal disc drives.

HP BASIC MSUS	MS-DOS Drive
":,1500,0"	A:
":,1500,1"	B:
":,1500,2"	C:
":,1500,3"	D:

Refer to appendix F if you want to change the standard configuration file.

Note 👆 The interrupt level on your language processor card must match the interrupt level in the configuration file. The default interrupt level on the card and in the file is "7." Information on changing the interrupt level on the card can be found in the HP 82321A *Language Processor Installation Instructions.* If you change the interrupt level on the card, you *must* change the interrupt level in the configuration file. The install procedure described later will do this for you.

Before You Start

Be sure that you have installed your language processor card. If you have not done so, refer to the *Language Processor Installation Instructions* that came with the card for installation instructions.

For best performance when using HP BASIC, you should set the number of MS-DOS file handles to a minimum of 20. You can use the MS-DOS configuration command "FILES=20" in your CONFIG.SYS file (refer to your MS-DOS user's reference manual for more information). You should also set the number of MS-DOS file buffers to a minimum of 10. You can do this with the MS-DOS configuration command "BUFFERS=10" in your CONFIG.SYS file. Be sure to reboot MS-DOS after changing your CONFIG.SYS file.

The following installation procedure will accomodate most allowable system configurations. Allowable configurations include all supported discs, display adapters, keyboards, built-in HP-HIL, serial or Centronics printers using LPT1, and SRM and GPIO interfaces. The language processor card interrupt level is set to 7, and the start address is 250H. If you need to change the interrupt level or the start address, you will have to change the configuration file. The install procedure described in the next section will do this for you.

Memory resident "hot key" programs such as SIDEKICK and SUPERKEY are not supported by HP BASIC.

Installing the HP BASIC Language Processor

The following procedures show you how to install your HP BASIC Language Processor on a high-capacity (1.2 Mbyte or greater) floppy disc or a hard disc. The process of installing involves copying files from your HP BASIC Language Processor discs onto a single high-capacity floppy disc or a hard disc. The purpose of this is to locate all of the files necessary to boot your HP BASIC Language Processor system from a single high-capacity floppy disc or a hard disc directory. Use the procedure which applies to your system, then continue with the section entitled "After HP BASIC Is Loaded."

Installing On a High-Capacity Floppy Disc Drive

Use the following table to determine which drive to use as the source (the drive which you will install *from*), which drive to use as the *target* (the drive that you will install *to*), and whether to use the 5.25-inch HP BASIC Language Processor discs or the 3.5-inch HP BASIC Language Processor disc when you install HP BASIC.

If Drive A Is...	And Drive B Is...	Then Install ... From...To...
5.25-inch low-capacity	5.25-inch low-capacity	You cannot install HP BASIC. You must have *at least* one high-capacity floppy drive.
5.25-inch high-capacity	5.25-inch high-capacity	Either: (1) 5.25-inch from A (source) to B (target) or (2) 5.25-inch from B (source) to A (target).
5.25-inch high-capacity	5.25-inch low-capacity	5.25-inch from B (source) to A (target).
5.25-inch low-capacity	5.25-inch high-capacity	5.25-inch from A (source) to B (target).*
5.25-inch low-capacity	3.5-inch	5.25-inch from A (source) to B (target).
5.25-inch high-capacity	3.5-inch	Either: (1) 5.25-inch from A (source) to B (target) or (2) 3.5-inch from B (source) to A (target).
3.5-inch	5.25-inch low-capacity	5.25-inch from B (source) to A (target)*
3.5-inch	5.25-inch high-capacity	Either: (1) 5.25-inch from B (source) to A (target)* or (2) 3.5-inch from A (source) to B (target).*
3.5-inch	3.5-inch	Either: (1) 3.5-inch from A (source) to B (target) or (2) 3.5-inch from B (source) to A (target)

* Refer to the examples that follow.

To clarify this procedure, consider the following examples:

- You have a 5.25-inch low-capacity drive in drive A and a 5.25-inch high-capacity drive in drive B. You will install HP BASIC using the 5.25-inch HP BASIC Language Processor discs. These discs will be placed in drive A (source) and installed on a single 5.25-inch high-capacity disc in drive B (target).

- You have a 3.5-inch drive in drive A and a 5.25-inch low-capacity drive in drive B. You will install HP BASIC using the 5.25-inch HP BASIC Language Processor discs. These discs will be placed in drive B (source) and installed on a single 3.5-inch disc in drive A (target).

- You have a 3.5-inch drive in drive A and a 5.25-inch high-capacity drive in drive B. You can install HP BASIC using *either* the 5.25-inch HP BASIC Language Processor discs *or* the single 3.5-inch HP BASIC Language Processor disc.

  - If you install HP BASIC using the 5.25-inch HP BASIC Language Processor discs, these discs will be placed in drive B (source) and installed on a single 3.5-inch disc in drive A (target).

  - If you install HP BASIC using the 3.5-inch HP BASIC Language Processor disc, this disc will be placed in drive A (source) and installed on a single high-capacity 5.25-inch disc in drive B (target).

Determine the source and target drives you will use and proceed with the installation:

1. Use the MS-DOS FORMAT command to format the target disc. This will be your system disc. If you already have a formatted disc prepared, disregard this step. Insert the formatted disc in the target drive as determined from the table.

2. Insert HP BASIC Language Processor disc one in the source drive and make the source drive the current drive. For example, if you want to install from drive B (source) to your system disc in drive A (target), insert disc one in drive B and type:

```
B: (Enter)
```

3. Run the install utility to complete the installation:

```
INSTALL (Enter)
```

The INSTALL utility is necessary to set up the HP BASIC directory file on the target disc. It will also ask you if you have changed the default configuration for the interrupt level or the start address. After you answer the questions, INSTALL will configure the system.

Install will then continue with the BOOT process. The system will go through a self-test mode during which interfaces such as keyboard, graphics, and HP-IB are tested. A list of Series 200/300 interface part numbers and the select code for the corresponding HP BASIC Language Processor interface are then displayed, along with the available HP BASIC Language Processor memory.

If you accidently press a key after the boot process is started, the boot process will be interrupted. After a short time you will see:

`Searching for a system (ENTER To pause)`

at the bottom of the screen. Shortly thereafter, you will see `1B SYSTEM_BA5` in the upper right corner of the screen. Type `1B` and the system will continue the boot process.

When the BOOT process is completed, you will hear a beep. The HP BASIC system will then be built. At this point you will have the option of having all binaries loaded automatically (recommended if you have at least 1 Mbyte of RAM on your language processor card), or selecting the binaries you want to load. The version of HP BASIC that you just booted already includes the EDIT, CS80, CRTA, GPIO, and HP-IB binaries.

If your language processor card does not have optional memory installed (either the HP 82303A RAM Expansion Kit or the HP 82305A RAM Expansion Board), you will not be able to load all the binaries. Refer to appendix D for a list of the available binaries and the amount of memory each requires. If your application requires all the available binaries, then you must add additional memory to the language processor card.

If you want to load all the available binaries, press Ⓨ when you see the message:

`Do you want to load all the binaries (Y/N)?`

If you answered N to the question, INSTALL will prompt you for each of the binaries to be loaded. Press Ⓝ for those binaries you do *not* want loaded and Ⓨ for those binaries you do want loaded. When you see the message:

```
BASIC is now loaded.
```

the procedure is complete. You can now go on to the section entitled "After HP BASIC Is Loaded."

Installing On a Hard Disc Drive

1. Insert HP BASIC Language Processor disc one in the source drive and make the source drive the current drive. For example, if you want to install from drive A (source) to your hard disc, insert disc one in drive A and type:

```
A: (Enter)
```

2. Run the install utility to complete the installation:

```
INSTALL (Enter)
```

The INSTALL utility is necessary to set up the HP BASIC directory file on the target disc. It will also ask you if you have changed the default configuration for the interrupt level or the start address. After you answer the questions, INSTALL will configure the system.

INSTALL will then continue with the BOOT process. The system will go through a self-test mode during which interfaces such as keyboard, graphics, and HP-IB are tested. A list of Series 200/300 interface part numbers and the select code for the corresponding HP BASIC Language Processor interface are then displayed, along with the available HP BASIC Language Processor memory.

If you accidently press a key after the boot process is started, the boot process will be interrupted. After a short time you will see:

```
Searching for a system (ENTER To pause)
```

at the bottom of the screen. Shortly thereafter, you will see 1B SYSTEM_BA5 in the upper right corner of the screen. Type 1B and the system will continue the boot process.

When the BOOT process is completed, you will hear a beep. The HP BASIC system will then be built. At this point you will have the option of having all binaries loaded automatically (recommended if you have at least 1 Mbyte of RAM on your language processor card), or selecting the binaries you want to load. The version of HP BASIC that you just booted already includes the EDIT, CS80, CRTA, GPIO, and HP-IB binaries. If you want to load all the available binaries, press Ⓨ when you see the message:

```
Do you want to load all the binaries (Y/N)?
```

If you answered N to the question, INSTALL will prompt you for each of the binaries to be loaded. Press Ⓝ for those binaries you do *not* want loaded and Ⓨ for those binaries you do want loaded. When you see the message:

```
BASIC is now loaded.
```

the procedure is complete.

After HP BASIC Is Loaded

Your HP BASIC system is now installed, booted, customized, and ready for use. You will find a complete list of keyboard functions in appendix E. An abbreviated list appears on the Key Function and Switch Configuration Guide.

You can exit HP BASIC at any time by pressing EXIT (Ctrl F10). You then return to HP BASIC by typing:

```
BASIC  Enter
```

To activate HP BASIC after you turn on your computer, follow these steps:

1. If you are using a high-capacity floppy disc, insert your system disc in the high-capacity drive.

2. Be sure your current disc drive and directory in MS-DOS is the one which contains your HP BASIC system.

3. Type BASIC Enter.

When the HP BASIC screen appears, your HP BASIC system is ready for use. An explanation of the HP BASIC screen is shown in the following figure and table.

The figure shows a display screen with the following labeled areas from top to bottom:
- Output Area
- Blank Line
- Display Line
- Keyboard Area (two lines)
- Message Results Line
- Softkey Labels (two lines)

With indicators (1), (2), (3) positioned above the Message Results Line.
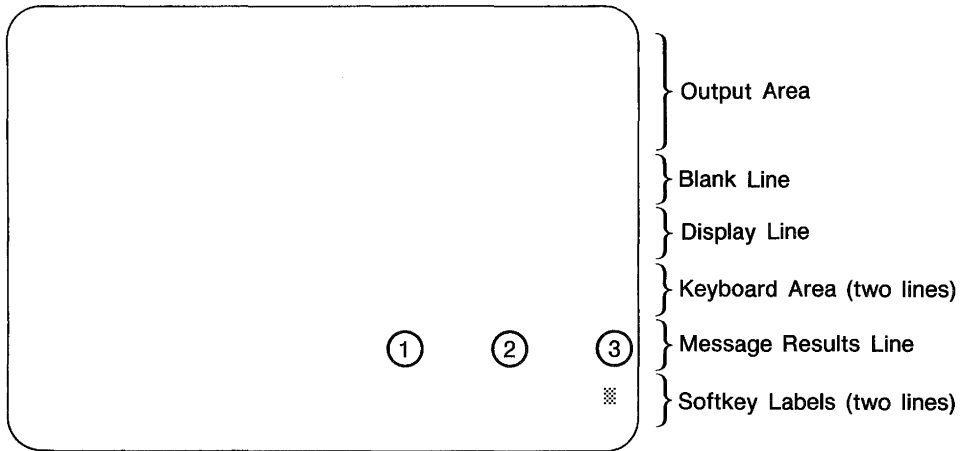
Label	Description
OUTPUT/PRINT	The largest portion of the display where characters specified in PRINT and OUTPUT statements are displayed when PRINTER IS CRT is active.
	System messages when PRINTER IS CRT and PRINTALL ON are active.
Disp	Destination of characters specifed in DISP statements.
Keyboard Input	Characters typed on the keyboard appear in this line.
System Messages	System messages and system status appear in the line.
Softkey labels	Labels of softkeys appear here if KEY LABELS is active.
(1) Softkey menu indicator	Select menu with SYSTEM or MENU key.
(2) Caps lock indicator	Toggle caps lock with (Caps Lock) key.
(3) Program status indicator	Indicates status (running, paused, idle) of program.

Your HP BASIC system can be customized to save language processor RAM by using the LIST BIN, LOAD BIN, and STORE SYSTEM statements.

- LIST BIN lists the binaries currently loaded in your system.
- LOAD BIN <file name> loads the specified binary into your system.
- STORE SYSTEM <file-name> stores the BASIC system into the specified file.

Testing Mass Storage Devices

You've already done this in the preceding steps. However, if you haven't tried all of your disc drives yet, here is a simple procedure to test them:

1. Insert an initialized disc into the drive. Remove all other discs from your other drives.

2. Use the syntax:

```
MASS STORAGE IS ":msus" (Enter)
```

to specify the drive that you are testing.

The following table shows the correlation between HP BASIC Mass Storage Unit Specifier (MSUS) and MS-DOS (Drive ID) for the default configuration for internal disc drives.

HP BASIC MSUS	MS-DOS Drive
":,1500,0"	A:
":,1500,1"	B:
":,1500,2"	C:
":,1500,3"	D:

To execute this statement, substitute the MSUS of your drive in the statement above. If you're using the B drive you would type:

```
MASS STORAGE IS ":,1500,1" (Enter)
```

3. Type:

```
CAT (Enter)
```

If the drive's access light flashes and information is listed on the screen, your drive is working normally.

Refer to the section entitled "The Configuration File" for more information.

Testing Printers

1. Turn on the printer.

2. Use the syntax:

```
PRINTER IS device select code (Enter)
```

substituting the device select code specific to your device.

To execute this statement, substitute your printer's device select number in the statement. For example if you are using a GPIO (Centronics) printer, type:

```
PRINTER IS 26 (Enter)
```

If you want to use a printer connected to the HP-IB port on your HP BASIC Language Processor card, type:

```
PRINTER IS 701 (Enter)
```

This table shows the device select codes for different devices.

Printer	Device Select Code
HP-IB	7
LPT1	26

3. Type:

```
PRINT "HELLO" (Enter)
```

Note 👉 Some printers require a full page of print to page advance, so use the manual form feed to advance the page or print a whole page of "HELLO".

If "HELLO" has printed the printer is working normally.

Testing Plotters

1. Turn the plotter on.
2. If you did not load the GRAPH binary into your custom system, do it now. Use LOAD BIN "GRAPH" and STORE SYSTEM to add this binary to your system.
3. Use the syntax:

```
PLOTTER IS device select code "HPGL" (Enter)
```

substituting your device select code to specify the plotter as the default plotter.

To execute this statement, substitute your plotter's device select code in the statement. If you have an HP-IB plotter, type:

```
PLOTTER IS 705,"HPGL" (Enter)
```

4. Type:

```
PEN 1 (Enter)
```

If the plotter picks up a pen, the plotter is working normally.

5. To replace the pen, type:

```
PEN 0 (Enter)
```

Testing the HP-HIL Mouse

1. Type: TIMEDATE (Do not press (Enter)).

2. If the mouse moves the cursor left and right when moved, it is working properly. Pressing either of the buttons on the mouse will be treated the same as if you pressed (Enter), and the TIMEDATE value will be displayed.

Now that the software installation is complete you're ready to start productively using HP BASIC. The next chapters contain instructions for programming, graphics, and interfacing. You may want to read the introduction at the beginning of each chapter to familiarize yourself with the possiblities of HP BASIC. Then just turn to the section you've selected to continue learning how to use HP BASIC.

3

Programming Techniques

Contents

3

Programming Techniques

Introduction

This chapter will introduce you to the BASIC 5.0 programming language and provide some helpful hints on how you can obtain the most from it. You do not need a high skill level in BASIC, but we assume you have some previous programming experience. If you have never programmed a computer before, it will probably be easier for you to start with one of the many beginner's text books available from various publishing companies. If you have experience on other HP desktop computer systems or with other high-level languages, you should find these programming procedures familiar. Whatever your starting point, it makes sense to learn the mechanics of program writing before you become absorbed in a study of all the program statements.

Explanation of Terms

Before proceeding, you should understand some common terms used in BASIC programming. This section will explain the meaning of some of the more frequently used terms.

Keyword. A keyword is a group of uppercase characters that is understood by the BASIC language system to represent some predefined action.

Statement. A statement is a keyword (sometimes optional) followed by any parameters, lists, specifiers, and secondary keywords that are allowed with that keyword.

Program Line. A program line contains at least a line number followed by a statement. It may also contain a line label, a name that is placed after the line number and terminated by a colon.

Program. A list of program lines, with an END statement on the last line.

Command. A command is a statement that is typed without a line number and executed. There are some commands that cannot be stored as program lines, such as DEL and SCRATCH. There are also some statements that cannot be executed as commands, such as DIM and RETURN.

Enter. Entering a program line means that you type a line number followed by a valid statement and then press the (Enter) key. The line is stored in memory as part of a program, but it performs no function until you run the program.

Note	The (Enter) key may appear on the keyboard as (Return), (Execute), (Exec), or (End Line), depending on the particular computer you are using. When you see (Enter) in this guide, use the key that corresponds to it on your keyboard.

Execute. Execute means that you type a statement with no line number and press (Enter). The command is executed immediately and is not stored in a program.

Using the Keyboard

In this chapter you will find references to keyboard *functions* rather than explicit keystrokes. This is because HP BASIC supports both the Vectra PC keyboard and the Enhanced Vectra PC keyboard. Refer to appendix E or the Key Function and Switch Configuration Guide for your keyboard to determine the keystrokes to use.

Using the Editor

The BASIC editor is a very versatile feature of the BASIC system. The following sections will show you how to use it properly.

Entering a Program

To enter a program into the computer you must be in the EDIT mode. You access EDIT by typing:

```
EDIT <line number>,<increment>
```

and pressing (Enter) or by pressing EDIT. If no parameters are present, EDIT assumes a line number of 10 and an increment between lines of 10. Once in EDIT, the format of the screen display is changed as shown in the following diagram.

Format of EDIT Screen

} Previous Program Lines (if any)

} Current Program Line (2 CRT lines)

} System Message Line (if needed)

} Following Program Lines (if any)

} Softkey Labels
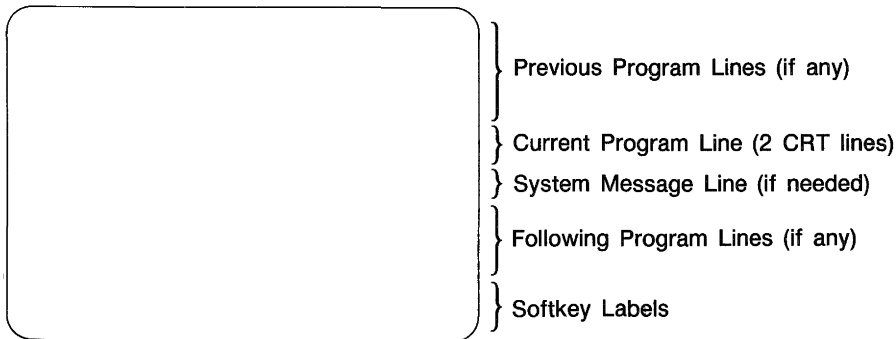
You can view several lines before and after the line you are editing. The system supplies the line number for the current line, and program portions can be viewed by simple scrolling.

Entering Program Lines. You enter program lines by typing them after the line number and pressing (Enter). The computer checks for syntax errors and converts letter case to the required form for names and keywords, and then stores the line. The computer supplies a line number automatically. If you want to change the line number, simply back up the cursor to the appropriate position and type in the line number you want. Changing the line number causes a copy operation, not a move. The original line still exists.

Inserting Lines. You can insert new lines between existing lines very easily. For example, assume you want to insert some lines between line 90 and line 100. Place line 100 in the current-line position and press INSERT LINE*. The program display "opens" and a new line number appears between line 90 and line 100. Type and store the inserted lines in the normal manner. The computer maintains the established increment between line numbers whenever possible. When the normal increment cannot be maintained, an increment of one is used. When there are no line numbers available between the current line and the next line, enough of the program below the current line is renumbered to allow the insert operation to continue.

Deleting and Recalling Lines. You can delete lines one at a time or in blocks. To delete the current line, press DELETE LINE*. If you delete a line by mistake, the line can be recovered by pressing the RECALL* function. To do this, use the following procedure:

1. Position the cursor below the line where you want to insert the deleted line.

2. Press INSERT LINE*. The program display will "open" and a new line number will appear.

3. Press RECALL*. The deleted line will appear.

4. Press (Enter). A new line number will appear beneath the line just recalled.

5. You can enter new lines at this point or move to another area of the program for other editing.

You can use the DEL command to delete lines. When DEL is followed by a single identifier, only that line is deleted. The identifier can be a line number or a label. Blocks of program lines can be deleted by using two identifiers with the DEL command. The first identifier identifies the first line of the block to be deleted, and the second identifier identifies the last line of the block. Here are some examples:

---

* Refer to appendix E or the Key Function and Switch Configuration Guide for the keystrokes to use.

```
DEL 100,200
```

Deletes lines 100 through 200, inclusive

```
DEL BLOCK2,32766
```

Deletes all lines from the one labeled "BLOCK2" to the end of the program.

```
DEL 250,10
```

Illegal because the line identifiers are not in order.

Renumbering a Program

You can renumber a program by using the REN command. You specify the starting line number, the interval between lines, and the range of lines. For example,

```
REN 100,5 IN 1,500
```

renumbers current lines 1 thru 500 using 100 for the first line number and an increment of 5 between line numbers. If the increment is not specified, 10 is assumed. If a range is not specified, the entire program is renumbered. When no parameters at all are specified, the computer assumes 10 for the first number and renumbers the entire program with an increment of 10.

Listing a Program

You can display or list all or part of your program by executing a LIST statement. The LIST statement has parameters that allow you to specify both the range of lines to be listed and the device to which the listing should be sent. If LIST is executed without any parameters, the default action is to list the entire program on the system printer. The default system printer after a power-on or SCRATCH A is the CRT. The system printer is defined by the PRINTER IS statement.

You specify starting and ending line numbers in the LIST statement, or you may specify labels instead. For example:

```
LIST 100,200
```

Lists lines 100 through 200, inclusive

```
LIST 1850
```

Lists the program from line 1850 to the end.

```
LIST Rocket
```

Lists the program from the line labeled "Rocket" to the end.

If you want the listing to be printed on an external printer, you must use the PRINTER IS statement prior to the LIST statement:

```
PRINTER IS 26
```

To make the CRT the system printer again use:

```
PRINTER IS 1
```

You can also use the LIST statement to list a program on the printer and keep the CRT as the system printer:

```
LIST #701
LIST #26
```

This statement sends the entire program listing to an HP-IB printer (address 01) without changing the system printer selection.

Editing A Program

Some commands make it easy to do large amounts of program editing very quickly. Among these are commands to move blocks of text, copy blocks of text, replace occurrences of one string with another string, find occurrences of a string, cross-reference the program, and more.

Moving Program Segments. You can move blocks of text with the MOVELINES command. This command moves contiguous program lines from one location to another. For example, if you wish to move the code in a program that is located between lines 100–250 to a new location in the program beginning with line number 1000, type:

```
MOVELINES 100,250 to 1000
```

Or, you could specify the lines by using labels:

```
MOVELINES label1,label2 TO new_block
```

If you intend to create a subprogram or function by moving a block of code, enter the subprogram header **before** moving the code. You cannot enter a SUB or DEF FN statement if there are other statements following it.

If the starting line number does not exist, the next line is used. If the ending line number does not exist, the previous line number is used. If a line label doesn't exist, an error occurs and no moving takes place. If an error occurs during the MOVELINE operation (a memory overflow, for example), the move is terminated and the program is left partially modified.

Copying Program Segments. The COPYLINES command performs the same function as MOVELINES, except that it leaves the code in the old location. This is desirable when you want a section of code that is very similar, but not identical to a section of code you already have. (If it were identical, you would probably put it into a subprogram.) It is often easier to copy code and modify one version than to type two separate, only slightly different versions.

Search and Replace Operations

The FIND Command. You can find all the occurrences of a particular string in a program by using the FIND command. When a program line that contains the specified string has been found, the computer places you in EDIT mode automatically. The current line is the line containing the specified string, and the cursor is positioned on the first character of the string. The message "Found 'string value'" is displayed in the system message line. You can then edit the string as you desire. When you press (Enter), the computer will continue its search for the string, stopping when it finds another occurrence of the string, when it reaches the end of the program, or when it reaches the last line of the specified range. To cancel a search operation before it is finished, press CLEAR I/O. The following examples illustrate the use of the FIND command.

```
FIND "STRINGA"
```

Searches for the first occurrence of the string "STRINGA", starting from the current location in the program.

```
FIND "STRINGB" IN 1500
```

Searches the program for the string "STRINGB" beginning at line 1500.

```
FIND "STRINGC" IN 1550, 1700
```

Searches the program for the string "STRINGC" beginning at line 1550 and ending at line 1700.

You can use line labels instead of line numbers if you wish.

The CHANGE Command. You can replace any string with another string by using the CHANGE command. CHANGE is like FIND in that it looks through your program and finds occurrences of the specified string. However it also makes a tentative change that you can confirm by pressing (Enter), or deny by pressing CONT. If you are positive that you don't need to verify each replacement, appending ;ALL to the CHANGE command will cause the search-and-replace to be done with no further action on your part.

```
CHANGE "OLD TEXT" TO "NEW TEXT"
```

The computer searches the entire program from the beginning and stops at any point where it finds the string "OLD TEXT". You are then asked the following question: "OLD TEXT" to "NEW TEXT?". Press (Enter) if you want the change made, or press CONT if you do not. In either case the computer will continue the search, repeating the above process whenever it finds the specified string.

```
CHANGE "OLD TEXT" to "NEW TEXT" IN 2600,3000
```

This performs exactly the same function as the previous command, except that the computer will only perform the search from program line 2600 through program line 3000.

```
CHANGE "OLD TEXT" to "NEW TEXT";ALL
```

This performs the same function as the first command, except that no verification on your part is required. The computer automatically makes the requested change.

Getting Out of EDIT Mode

There are many ways to terminate the EDIT mode. If you want to return the CRT to its "normal" mode, press PAUSE or (CTRL) (Home) (CLEAR SCREEN).

Another way to terminate EDIT mode is to proceed with another operation by pressing the appropriate function key. Initiating operations such as LOAD, CAT, LIST, RUN, STEP, or PAUSE will automatically terminate EDIT mode.

Indenting

You can indent your program in appropriate places by using the INDENT command. This command automatically indents whenever there is the beginning or end of a program statement which causes looping, is conditionally executed, or is a separate program segment. There are two parameters, starting column number (default = 6) and increment (default = 2). The starting column number is the column in which the first character of the first statement of each context appears. The increment specifies the number of spaces that the beginning of the lines move to the left or right when the nesting level of the program changes.

Indenting a program may cause the length of some of the lines to become longer than the computer can list. This condition is indicated by the presence of an asterisk (*) after the line numbers of affected lines. If this occurs, the program will run properly, store properly, and load properly. However, you cannot do a SAVE, then a GET. Doing an INDENT with smaller values will alleviate this problem.

You can see the effect of the INDENT command from the following example. You can type in the program or read it from the flexible disc supplied with your BASIC system. If you choose the latter method, place the disc in drive A and type the following:

```
LOAD "INDNTPGM:CS80,1500,0"
```

Otherwise, enter the program as it is shown on the following page.

```
10 FOR I = 1 TO 5
20 REPEAT
30 INPUT "How old are you?",Age
40 Reasonable = 1
50 IF Age < 0 THEN
60 DISP "Forgive me, but you can't be ";Age;"years old."
70 Reasonable = 0
80 ELSE
90 IF Age >= 120 THEN
100 DISP "That's a little difficult to believe."
110 Reasonable = 0
120 ELSE
130 IF Age >= 100 THEN
140 DISP "You are getting up there, aren't you?"
150 ELSE
160 IF Age >= 60 THEN
170 DISP "I'm impressed.  You don't look that old."
180 ELSE
190 IF Age >= 40 THEN
200 DISP "Ah, you're over the hill."
210 ELSE
220 DISP "So, just a youngster."
230 END IF
240 END IF
250 END IF
260 END IF
270 END IF
280 WAIT 2
290 UNTIL Reasonable
300 DISP "You were";Age*365.242198781;"days old on your last birthday."
310 WAIT 2
320 NEXT I
330 END
```

Then type the command INDENT and press (Enter). List the program and you will see the results. The program should now look like this:

```
10    FOR I = 1 TO 5
20      REPEAT
30        INPUT "How old are you?",AGE
40        REASONABLE = 1
50        IF AGE < 0 THEN
60          DISP "Forgive me, but you can't be ";AGE;"years old."
70          REASONABLE = 0
80        ELSE
90          IF AGE >= 120 THEN
100           DISP "That's a little difficult to believe."
110           REASONABLE = 0
120         ELSE
130           IF AGE >= 100 THEN
140             DISP "You are getting up there, aren't you?"
150           ELSE
160             IF AGE >= 60 THEN
170               DISP "I'm impressed.  You don't look that old."
180             ELSE
190               IF AGE >= 40 THEN
200                 DISP "Ah, you're over the hill."
210               ELSE
220                 DISP "So, just a youngster."
230               END IF
240             END IF
250           END IF
260         END IF
270       END IF
280       WAIT 2
290     UNTIL REASONABLE
300     DISP "You were";Age*365.242198781;"days old on your last birthday."
310     WAIT 2
320   NEXT I
330   END
```

Running a Program

You run a program by pressing the RUN keys or by typing RUN and pressing (Enter). This tells the computer to go through a pre-run phase and then begin normal program execution with the lowest numbered line in the main program. The RUN command can also be followed by a line identifier that lets you specify where the program execution is to begin.

Program Execution

The process of program execution as implemented by the BASIC interpreter is summarized below.

1. Determine which program line is to be acted upon next.

2. Identify the statement that follows the line number and label (if any) on that line.

3. If the statement has a run-time action, perform that action.

4. Repeat steps 1 through 3 until an END, STOP, or PAUSE statement is executed.

The RUN command determines which line is acted on first. Executing RUN with no parameters, or pressing the RUN keys causes the execution process to begin at the first line of the program. Execution can be started anywhere in the program by using the RUN command with a line identifier. For example:

```
RUN 200
```

This command causes execution of the program to begin at line 200. If there is no line 200, execution begins with the line whose number is closest to and greater than 200. The line identifier can also be a label. For example:

```
RUN Spot_run
```

This command causes execution of the program beginning with the line labeled "Spot_run". If there is no such label, an error results.

Live Keyboard

The term "live keyboard" is used when talking about commands that are executed from the keyboard while a program is running. The keyboard is still active when a program is running. You can execute commands, change variables, and change the state of the computer.

Pausing and Stopping. If the operator does not intervene, a program will run until it encounters an END, PAUSE, or STOP statement. For example, if you wish to pause program execution before its normal completion, press PAUSE. This causes a temporary halt to program execution. To continue, press CONT. If you wish to stop the program, press STOP.

The "Run Light". You can determine the current state of the computer by the indicator in the lower-right hand corner of the CRT. The character in this corner is referred to as the "run light". The following table defines the various indications of the run light.

Status Indicator	Run Light	Computer State
Idle	blank	Program stopped; CONTINUE not allowed
Running	■	Program running
Paused	–	Program paused; may be continued
Transfer	I O	Program paused, but a TRANSFER is still active
Input	?	Computer is waiting for input from the keyboard
Command	*	Computer is executing a command from the keyboard

An Example. To demonstrate some of the interaction between a program and the keyboard, use the EDIT mode to enter the following program.

```
10 DISP "NEXT COMMAND?"
20 X=0
30 PRINT X;
40 X=X+1
50 WAIT .1
60 GOTO 30
70 END
```

1. After you have entered the program, run the program by pressing RUN. This will automatically get you out of EDIT mode and begin running the program.

2. Press PAUSE. The printout of numbers stops, and all data on the CRT remains unchanged. The run light indicates that the program is paused and can be continued. The program line that appears at the bottom of the CRT is the next line of the program that will be executed when program execution resumes.

3. Press STEP a number of times. The program is now executed one line at a time, as indicated by the program lines changing at the bottom of the screen. Notice that the program is still paused and continuable after each press of the STEP keys.

4. Press CONT. The printout on the CRT resumes with the next number in sequence, and the run light indicates the program is running.

5. Press STOP. The printout of numbers stops, and all the data on the CRT remains unchanged. The run light is off.

6. Press CONT. An error results because a stopped program cannot be continued.

7. Press RUN. The program runs again, but the number sequence has started from the beginning. RUN causes the program to start from the beginning, not resume.

8. Type x = 0 and press (Enter). Notice that the numbers being printed start over from "1". The live keyboard was used to change the value of "X", and the program used this new value immediately.

9. Type WAIT 5 and press (Enter). Notice that the run light changes to indicate that a keyboard command is being executed. The printout is delayed for five seconds while the keyboard command is processed.

10. Press PAUSE, and then type EDIT 50 and press (Enter). The display on the CRT changes to show the program, and line 50 appears in the current-line position of the screen. The run light indicates that the program is paused.

11. Change line 50 to WAIT 2 and press (Enter). The new line 50 is entered, but the run light goes out. Changing the program caused it to move from the paused state to the stopped state.

12. Press CONT. An error results. Once a program has been changed, the program is no longer paused, and the CONT command is not allowed.

Program Storage and Retrieval

The previous sections have shown you how to enter, edit, and run a program. The next logical step is to save the program for future use or further development.

The exact procedure for storing and retrieving programs depends upon the type of mass storage device you are using. Your computer may have an internal floppy disc drive, an internal hard drive, an SRM system, or one of the many external disc drives that are compatible with your system.

What Is Mass Storage?

As the adjective "mass" suggests, mass storage devices are data-storage devices which are generally capable of storing "large" amounts of data. Just how much data constitutes a large amount depends on the device itself. Most mass storage devices are capable of storing hundreds of thousands to several million items.

Besides having the ability to store data, mass storage devices are capable of providing means for keeping data organized so that logical groups may be accessed systematically and efficiently. Data items are organized into logical groups of data known as *files*; a file is merely a collection of data items. Mass storage *directories* are composed of one or more files. On some HP mass storage devices, a directory consists of all files on the mass storage *media*; mass storage media are the actual physical means by which data are stored. For example, the media used by the internal drive of your computer consists of magnetic particles on a plastic disc which can be magnetized to store data.

Media Specifiers

Once the mass storage is connected, you need a way of specifying which mass storage device is to be accessed. This is done with a media specifier. The syntax for a media specifier is illustrated below. Each component is then discussed.

```
MASS STORAGE IS ": [<device type>],<device
selector>[,<unit number>]"
```

Device type—effectively describes the mass storage device to the system. The system·then knows the capacity of the device, the directory structure, and other information required to determine the access method for the device. Examples are

```
MEMORY, CS80, and HP82901
```

Device type CS80 is used for internal drives. If the device type specified is not valid, the system tests the device to determine its type. There are two exceptions to this.

1. If the device selector is 0 and the device type is invalid, the device type is assumed to be MEMORY.
2. If the device type is valid and the driver binary for the device is not loaded, the system considers the device an invalid device type.

Device Selector—tells the system the select code of the interface connected to the device; if the interface is an HP-IB, it also tells the system the device's primary address. The system then knows which interface connects the device to the computer (and the device's address, if an HP-IB is used).

A device selector can be just an interface select code or a combination of select code and primary address. To derive a device selector with a primary address, multiply the interface select code by 100 and then add the address. For instance, the device selector 703 would select the device with primary address 3 which is connected to the interface at select code 7. Note that interface select code 7 is the built-in HP-IB interface; this is the interface you will probably use to attach external disc drives. The device selector for built-in drives is 1500.

Unit number—is used to select among the various built-in disc drives and directories. In the default configuration, A: is unit number 0, B: is 1, C: is 2, and D: is 3.

Examples. The following statements set the system mass storage to an HP 82901 drive at interface select code 7; the HP 82901 is set to primary address 0 and has a unit number of 1.

```
MASS STORAGE IS ":HP82901,700,1"
```

or

```
MASS STORAGE IS ":HP,700,1"
```

Note   MSI is a valid abbreviation for MASS STORAGE IS, and is easier to type.

Executing the following statement catalogs the disc in the internal drive at interface select code 1500 and unit number 0.

```
CAT ":CS80,1500,0"
```

The following statement creates an ASCII file named "Fred" on the disc in unit 3 of an HP 9134 drive, connected through interface select code 7; the device has a primary address of 0.

```
CREATE ASCII "Fred:HP9134,700,3"
```

Initializing a Disc

Before a disc is used for the first time, it must be initialized. If the disc has already been initialized, and it contains data you wish to retain, then it can be used on the BASIC system without initialization. However, if you don't need the data on a previously used disc, it might be advantageous to re-initialize it on your computer to get maximum performance. The point is this: a disc must be properly initialized before your computer can use it, but initializing a disc *destroys all the data on the disc.*

The following steps show a typical initialization process using an internal floppy disc. The procedure for initializing external discs is very similar, but specific details will change. For example, an external disc drive will have a different specifier, may have a different write-protect convention, and will probably take a different length of time to initialize.

This procedure will initialize the disc in MS-DOS format. Discs that have been initialized by using the MS-DOS FORMAT command can be used without going through this procedure. If you need to initialize a disc in LIF format for use on an HP series 200 or 300 computer, use the HPWUTIL program (F1 soft key). Refer to appendix C for more information.

To initialize a 5.25-inch disc on an internal disc drive A, follow these steps:

1. Make sure that the disc does not contain any important data or programs. Many types of computers and word processors use similar discs. *When a disc is initialized, all the data on it is destroyed!*

2. Ensure that the disc is not "write protected". The disc envelope has a small notch on one side. When this notch is open, the computer is allowed to write on the disc. If this notch is covered, data may be read from the disc, but recording is not allowed. Trying to initialize a write-protected disc results in error number 83.

3. Be sure the disc is properly inserted in the disc drive.

Note 👊 The next step assumes that you are using the default HP BA-SIC configuration or that drive A is the first drive specified in your HP BASIC configuration file.

4. Execute `INITIALIZE ":CS80,1500,0"`. This command tells the computer to erase all data from the disc, format it for use in your computer, check the quality of the media, and create the directory area.

An initialize operation takes about three minutes. The CRT displays the system's progress during this operation. When the initialization is complete, the message is displayed:

```
Formatting...Format complete.

1213952 bytes total disc space.
1213952 bytes available on disk.

Format another (Y/N?)
```

After the initialization has completed successfully, the disc is ready for storing programs and data.

Disc Labels

After you initialize the disc, you may want to give it a label. The PRINT LABEL statement prints the label in the disc directory. Once the label is there, a READ LABEL statement can retrieve it. The disc label is included in a CATalog of the disc.

For example, to give the disc in the A: disc drive the label VOL1, execute the following:

```
PRINT LABEL "VOL1" TO ":CS80,1500,0"
```

To read the label, enter:

```
READ LABEL Name$ FROM ":CS80,1500,0"
```

All mass storage operations, including program storage, require a properly initialized device. You can tell if a mass storage device has been properly initialized by executing a CAT command for that device. This command will display the contents of a device's directory. Type CAT and press (Enter). If the CRT displays a catalog listing, then you are looking at the directory of the default mass storage volume. Therefore, the device is properly initialized and can be used for program storage. If you get an Error 80, then there is no disc in the default drive, the disc has not been inserted properly, or the disc is write protected (if this is the first time the disc has been used). If you get an error, there are several things you might do, depending on your situation.

- Be sure the appropriate driver binaries have been loaded. Refer to chapter 2 for a description of loading binaries.
- If the error is caused by a disc that has not been initialized, or has been initialized improperly (typically errors 78, 84, or 85), you can execute an INITIALIZE command.

Caution

When you initialize a disc, all data on the disc is destroyed.

- Be sure that your mass storage system is configured properly.
- If you need further assistance, call your local Hewlett-Packard representative.

Recording a Program

To record a program, you can use the SAVE or STORE command with a suitable file name. The command used depends upon the type of file you want. If SAVE is used, the text of the program is recorded in an ASCII file. If STORE is used, the program is recorded in a PROG file. The main advantage of a PROG file is rapid access. The following table gives a brief summary of the differences between SAVE and STORE.

	SAVE	STORE
File type created:	ASCII	PROG
Retrieved by:	GET	LOAD
Can file be read as data?	Yes	No
Arbitrary program segments allowed?	Yes	No

To store a program, type the keyword STORE followed by a file name, and press (Enter). For example, the command to create a file called "mortgage" is:

```
STORE "mortgage"
```

The SAVE procedure is similar except that SAVE allows you to use line identifiers to specify what portion of the program you want to save. This is helpful when moving or appending program segments during major editing operations. To save all of a program in a file called "WHALES", execute the following command:

```
SAVE "WHALES"
```

The next command saves the last part of a program, from line 500 to the end, in an ASCII file called "TEMP".

```
SAVE "TEMP", 500
```

When both the starting and ending lines are specified, any portion of a program can be saved. Executing the command

```
SAVE "sort_code",Sort,Printout
```

saves that portion of the program that is between the lines labeled "Sort" and "Printout" (inclusive) in an ASCII file called "sort_code".

You can only use SAVE and STORE when first recording a file. If the file you are trying to use already exists, you will get an error message. To save or store a program to an existing file, you must use RE-SAVE or RE-STORE.

Retrieving a Program

Programs saved in an ASCII file are retrieved with the GET statement. Programs stored in a PROG file are retrieved with a LOAD statement. These statements can be executed from the keyboard as commands or included in a program. To retrieve a program you need to know the name and type of the file in which it is stored. If you are not sure of either of these, use the CAT comand. The catalog display shows the name and type of all files on the disc.

Using GET as a Command. You can use the GET command to bring in programs or program segments from an ASCII file, with the options of appending them to an existing program or beginning program execution at a specified line.

If you want to clear any existing program from memory and bring in the contents of an ASCII file, type:

```
GET "FORMULA"
```

This command clears the computer's memory and brings in the ASCII file called "FORMULA". If the first line of the file is not a valid program line, the GET is not performed and an error 68 is reported. If the file is not an ASCII file, the GET is not performed and an error 58 is reported.

If you want to append the contents of an ASCII file to an existing program, a line identifier is added to the GET command. For example, assume you have a program in memory whose last line number is 740, and you want to append the contents of a file called "George". You can use the following command to accomplish this:

```
GET "George",750
```

This appends the program lines from the file called "George" to the existing program, renumbering them to start with line number 750.

If the specified renumbering would create an invalid line number, an error is sent to the system printer with an error message, but it is not entered into program memory.

The GET command can also specify that program execution is to begin. This is done by adding two line identifiers: one specifies the placement and renumbering just described, and the other specifies the line at which execution is to begin. For example, assume there is no program in memory and that an ASCII file called "RATES" contains valid program lines. A typical command to bring a program into memory and begin execution at the first line is:

```
GET "RATES",10,10
```

If there is already a program in memory, an append run is allowed. For example:

```
GET "RATES",250,100
```

This command specifies that any existing lines from 250 to the end are to be deleted, the contents of "RATES" is to be renumbered and appended beginning at line 250, and then program execution is to begin at line 100.

Using GET in a Program Line. The GET statement can be used in a program to transfer execution from one program segment to another. This example of a programmed GET demonstrates a simple linkage of two program segments, as might occur when the entire program is too large to fit in available memory.

First Program Segment:

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

File WATTAGE:

```
10 COM Ohms,Amps,Volts
20 Watts = Amps*Volts
30 PRINT "Resistor Ohms =";Ohms
40 PRINT "Resistor Wattage =";Watts
50 END
```

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

Using LOAD as a Command. The LOAD command is used to bring in programs from a PROG file, with the option of beginning program execution at a specified line. For example:

```
LOAD "CANNON"
```

This command clears memory and loads the contents of the PROG file called "CANNON". If the file is not a PROG file, the LOAD is not performed and an error 58 is reported. If any lines require a language extension that is not currently installed, those lines cannot be executed. However, the LOAD proceeds without error.

The LOAD command can also specify that program execution is to begin.

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

This command causes the computer to load the program in file "STONE" and begin execution at line 10. The line identifier may be a label or a line number, but it must identify a line in the main program segment, not in a SUB or user-defined function.

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

Using LOAD in a Program Line. When used in a program line, the actions of the LOAD statement are the same as those described for the LOAD command, except program execution resumes whether a line identifier is specified or not. For example:

```
120 LOAD "PART2"
```

When this program statement is executed, the existing program is replaced by the contents of the PROG file called "PART2", and program execution resumes with the first line in the new program.

Program Structure and Flow

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

Sequence

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

- Linear flow involves no decision making.
- Linear flow is the default mode of execution. Unless you include a statement that stops or alters program flow, the computer will always "fall through" to the next higher numbered line after finishing the line it is on.

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

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

2. The STOP statement. This acts just like an END statement in that it stops program flow. You use a STOP statement when you desire program flow to stop at some point other than the end of the main program.

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

To demonstrate, type in the following program:

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

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

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

1. The GOTO statement causes the program to branch to some line number or label that is not the next line in the program. Following are examples of the GOTO statement:

```
100 GOTO 30
150 GOTO XXXX
      .
      .
300 XXXX:.........
```

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

```
100 GOSUB 1000

450 GOSUB 3000
```

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

Selection

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

1. Conditional execution of one segment.

2. Conditionally choosing one of two segments.

3. Conditionally choosing one of many segments.

Conditional Execution of One Segment. The basic decision to execute or not execute a program segment is made by the IF...THEN statement. This statement includes an expression that is evaluated as being either true or false. If true, the conditional segment is executed. If false, the conditional segment is bypassed. The conditional segment can be either a single BASIC statement or a program segment containing any number of statements. The following example shows conditional execution of a single statement:

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

Notice the test (Ph > 7.7) and the conditional statement (PRINT...) which appear on either side of the keyword THEN. If the value of Ph is greater than 7.7 the PRINT statement is executed. If the value of Ph is equal to or less than 7.7 the PRINT statement is not executed. In either case, the line number immediately following line 100 would be executed next.

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

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

The statement checks the value of a variable called Free_space, and if it is less than 100, a subroutine called Expand_file is executed. If the value is not less than 100, the subroutine is not executed. One important feature of this structure is that the program flow is essentially linear, except for the conditional "side trip" to a subroutine and back.

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

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

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

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

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

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

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

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

```
Flag = 1                                    Flag = 0

      400    IF Flag THEN
      410       R=R+2
      420       Area=PI*R^2
      430    ELSE
      440       Width=Width+1
      450       Length=Length+1
      460       Area=Width*Length
      470    END IF
      480    PRINT "Area =";Area
      490    ! Program continues
```
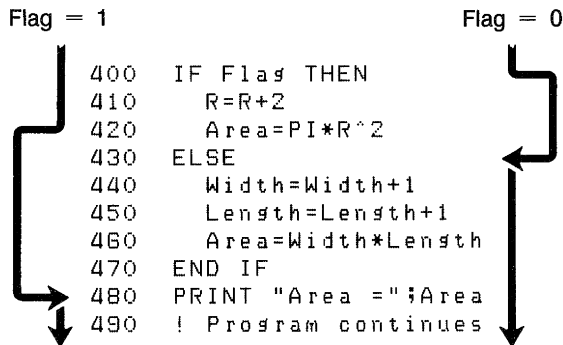
This example has an IF...THEN...ELSE structure which makes the one-of-two choice easy and readable.

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

Consider the processing of readings from a voltmeter. Readings which contain a function code have been taken. The function codes identify the type of reading and are shown in the following table:

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

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

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

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

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

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

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

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

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

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

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

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

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

Repetition

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

Fixed Number of Iterations. The general concept of repetitive program flow can be shown with the FOR...NEXT structure. The FOR statement marks the beginning of the repeated segment and establishes the number of repetitions. The NEXT statement marks the end of the repeated segment. This structure uses a numeric variable as a loop counter. The following example shows the basic elements of a FOR...NEXT loop:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Arbitrary Exit Points. The loop structures discussed so far do not allow for conditional exit points within the program segment between the top and bottom of the loop. The LOOP...EXIT IF construct allows you to do this. It also allows you to have more than one exit point. Also, the EXIT IF statement can be at the top or bottom of the loop. This means that the LOOP structure can serve the same purposes as the REPEAT...UNTIL and WHILE...END WHILE.

The EXIT IF statement must appear at the same nesting level as the LOOP statement for a given loop. The following two examples demonstrate this.

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

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

Here is the proper structure to use.

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

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

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

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

Event-Initiated Branching

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

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

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

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

```
100 ON KEY 1 LABEL "Inc" GOSUB PLUS
110 ON KEY 5 LABEL "Dec" GOSUB MINUS
120 !
130 SPIN: DISP X
140   GOTO SPIN
150 !
160 PLUS: X = X + 1
170   RETURN
180 !
190 MINUS: X = X - 1
200   RETURN
210 END
```

The ON KEY statements are executed only once at the start of the program. Once defined, these event-initiated branches remain in effect for the rest of the program. The program segment labeled "SPIN" is an infinite loop. If it weren't for interrupts, this program couldn't do anything except display a zero. However, there is an implied IF..THEN at the end of lines 130 and 140 because of the ON KEY action. Either the "PLUS" or "MINUS" subroutines are selected as a result of softkey presses. If no softkey is pressed, the computer continues to display the value of X. The following section of pseudocode shows the program flow of the "SPIN" segment looks like.

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

Note that the only way to terminate this program is to type STOP and press (Enter).

Numeric Computation

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

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

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

Numeric Data Types

There are three numeric data types in BASIC:

- COMPLEX.
- INTEGER.
- REAL.

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

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

a + ib

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

INTEGER Variables. An INTEGER variable can be any whole-number value from -32768 through $+32767$.

REAL Variables. A REAL variable can be any value from $-1.797073134862315 \times 10^{-308}$ through $1.797073134862315 \times 10^{-308}$. The smallest non-zero REAL value allowed is approximately $\pm 2.225073858507202 \times 10^{-308}$

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

```
COMPLEX B, C, Phasor1(10), Phasor2(10)
INTEGER I, J, Days(5), Weeks(5:17)
REAL X, Y, Voltage(4), Hours(5,8:13)
```

each declare two scalar and two array variables. A scalar is a variable which can represent a single value. An array is a subscripted variable, and can contain multiple values accessed by subscripts. You can specify both the lower and upper bounds of an array, or specify the upper bound only, and use the existing OPTION BASE statement as the lower bound.You may declare an array using the DIM statement:

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

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

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

The ALLOCATE statement allows you to dynamically allocate memory in programs which need tight control over memory use. Arrays will be discussed in detail later in this section.

Type Conversions. The computer will automatically convert between REAL and INTEGER values in assignment statements and when parameters are passed by value in program and function calls. When parameters are passed by reference, the conversion will not be made, and a TYPE MISMATCH error will be reported.

When a REAL number is converted to an INTEGER, the fractional part is lost, and the REAL number is rounded to the closest INTEGER value. Converting the number back to REAL will not restore the fractional part. Also, because of the difference in ranges between the two types, not all REAL values can be converted into an equivalent INTEGER value. This problem can generate INTEGER OVERFLOW errors. The rounding problem does not generate an execution error, but the range problem can generate an execution error, and you should protect yourself from this possibility. One way to do this is shown on the next page:

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

Resident Numerical Functions

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

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

Arithmetic Functions. Your BASIC language has the following arithmetic functions included:

Function	Description
ABS	Returns the absolute value of an expression.
FRACT	Returns the fractional part of the argument.
INT	Returns the greatest integer that is less than or equal to an expression. The result is of the same type as the original number.
PI	Returns the constant 3.14159265358979, an approximate value for π.
SGN	Returns the sign of an expression: 1 if positive, 0 if 0, -1 if negative.
SQR	Returns the square root of an expression.

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

Function	Description
EXP	Raise the Naperian e to a power. e $\approx$ 2.71828182845905.
LGT	Returns the base 10 logarithm of the expression.
LOG	Returns the natural (Naperian base e) logarithm of an expression.

Trigonometric Functions. There are twelve trigonometric functions included in your BASIC language. All of these functions use radian values as the default input, however you can change this to degrees with the DEG statement. You can reselect radians by using the RAD statement. It is a good idea to explicitly set the mode for input to these functions, even if you are using the default (radian) mode. This is especially important when you are writing subprograms, as the subprogram inherits the mode from the calling program.

The following is a list of the trigonometric functions. All these functions use INTEGER, REAL, or COMPLEX numbers as their argument.

Function	Description
ACS	Returns the arc cosine of an expression.
ACSH	Returns the hyperbolic arc cosine of an expression.
ASN	Returns the arc sine of an expression.
ASNH	Returns the hyperbolic arc sine of an expression.
ATN	Returns the arc tangent of an expression.
ATNH	Returns the hyperbolic arc tangent of an expression.
COS	Returns the cosine of the angle represented by the expression.
COSH	Returns the hyperbolic cosine of the angle represented by the expression.
SIN	Returns the sine of the angle represented by the expression.
SINH	Returns the hyperbolic sine of the angle represented by the expression.
TAN	Returns the tangent of the angle represented by the expression.
TANH	Returns the hyperbolic tangent of the angle represented by the expression.

Binary Functions. All computer operations use the binary number representation. You usually don't see this because the computer changes decimal numbers that you input into binary representation. The operations you specify are performed on the binary numbers, and results are changed back into decimal numbers before displaying or printing them.

The following BASIC functions deal with binary numbers:

Function	Description
BINAND	Returns the bit-by-bit logical and of two arguments.
BINCMP	Returns the bit-by-bit complement of two arguments.
BINEOR	Returns the bit-by-bit exclusive or of two arguments.
BINIOR	Returns the bit-by-bit inclusive or of two arguments.
BIT	Returns the state of a bit of the argument.
ROTATE	Returns a value obtained by shifting an integer representation of an argument a specific number of bit positions with wraparound.
SHIFT	Returns a value obtained by shifting an integer representation of an argument a specific number of bit positions, without wraparound.

When any of these operations are used, the arguments are first converted to integer (if they are not already in integer) and then the specified operation is performed. You should restrict bit-oriented binary operations to declared INTEGER variables. If it is necessary to operate on REAL variables, be sure to use the precautions described under type conversions in the previous section, to avoid INTEGER OVERFLOW errors.

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

Function	Description
MAX	Returns the larger of a list of expressions.
MAXREAL	Returns the largest REAL number.
MIN	Returns the smallest of a list of expressions.
MINREAL	Returns the smallest REAL number.

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

Function	Description
DROUND	Rounds a number to a specified number of digits.
PROUND	Returns the value of the argument rounded to a power of ten.

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

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

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

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

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

Complex Functions. These functions are obtained by loading the COMPLEX binary, as described in chapter 2. Topics which are covered in this section are:

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

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

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

Executing the above program produces these results:

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

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

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

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

```
0    1
```

where 0 is the real part of the complex number and 1 is the imaginary part of that same number.

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

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

The results from executing this program are as follows:

```
.5 (degree mode)
```

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

```
.5 (degree mode)
```

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

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

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

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

Executing this statement produces:

```
10
```

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

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

Executing this statement produces:

```
-3
```

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

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

Executing this statement produces the following results:

```
10 3
```

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

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

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

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

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

If you change line 140 above to be:

```
140 DEG
```

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

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

Time and Date Functions. There are two functions which will return the time and date in seconds. These are:

Function	Description
DATE	Converts a formatted date string ("DD MMM YYYY") into a numeric value in seconds.
TIME	Converts a formatted time-of-day ("HH:MM:SS") string into a numeric value of seconds since midnight.

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

Function	Description
DVAL	Returns the whole number value of a binary, octal, decimal, or hexadecimal 32-bit integer. The argument is a a string.
IVAL	Returns the integer value of a binary, octal, decimal, or hexadecimal 16-bit integer. The argument is a string.

General Functions. When you are specifying select code and device selector numbers, it is more descriptive to use a function to represent that device as opposed to a numeric value. For example, the statement

```
ENTER 2;Numeric_function
```

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

```
ENTER KBD;Numeric_value
```

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

Function	Description
CRT	Returns the INTEGER 1. This is the select code of the internal CRT.
KBD	Returns the INTEGER 2. This is the select code of the keyboard.
SC	Returns the interface select code associated with an I/O path name.
PRT	Returns the INTEGER 701. This is the default (factory set) device selector for an external HP-IB printer.
RES	Returns the last live keyboard numeric result.

Array and Matrix Functions. The following functions perform operations connected with arrays or matrices.

Function	Description
BASE	Returns the lower subscript bound of a dimension of an array.
DET	Returns the determinant of a matrix.
DOT	Returns the inner (dot) product of two vectors.
RANK	Returns the number of dimensions in an array.
SIZE	Returns the number of elements in a dimension of an array.
SUM	Returns the sum of all the elements in an array.

Evaluating Scalar Expressions

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

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

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

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

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

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

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

1. Perform the calculations inside the parentheses and multiply by 4.

2. Compute the sine of Y.

3. Multiply the sine of Y by 5.

4. Add the value found in step 1 to the value found in step 3.

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

Step Functions. The comparison operators are useful for conditional branching, but you can also use them for creating numeric expressions representing step functions. For example, suppose you want to output certain values depending on the value, or range of values, of a single variable. This is shown as follows:

■ If variable < 0 then output $= 0$.

■ If $0 \leqslant$ variable < 1 then output $= \sqrt{A^2 + B^2}$.

■ If variable $\geqslant 1$ then output $= 15$.

You could achieve the desired result by using a series of IF..THEN statements, but you could also use the following expression (where X is the variable and Y is the output):

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

The boolean expressions each return a 1 or 0 which is then multiplied by the accompanying expression. Expressions not matching the selection return 0 and are not included in the result. The value assigned to the variable before the expression is evaluated is used to determine the result.

Comparing REAL Numbers. When you compare INTEGER numbers, no special precautions are necessary. When you compare REAL numbers, especially the results of calculations and functions, it is possible to encounter problems due to rounding. For example, consider the use of comparison operators in IF..THEN statements to check for equality in the following:

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

You will find that the equality test fails due to rounding errors. A repeating decimal or irrational number cannot be represented exactly in any finite machine.

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

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

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

Numeric Arrays

An array is a multi-dimensioned structure of variables that are given a common name. The array can have one through six dimensions. Each location in an array can contain one variable value, and each value has the characteristics of a single variable, depending on whether the array consists of REAL, INTEGER or COMPLEX values. A one-dimensional array consists of n elements, each identified by a single subscript. A two-dimensional array consists of m times n elements where m and n are the maximum number of elements in the two respective dimensions. Arrays require a subscript in each dimension in order to locate a given element of the array. You can specify up to six dimensions for any array in a program. REAL arrays require eight bytes of memory for each element, plus overhead, and COMPLEX arrays require 16 bytes of memory for each element, plus overhead. It is easy to see that large arrays can demand massive memory resources. An undeclared array is given as many dimensions as it has subscripts in its lowest-numbered occurrence. Each dimension of an undeclared array has an upper bound of ten. Space for these elements is reserved whether you use them or not.

Dimensioning an Array

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

```
COMPLEX Array_complex(2,4)
```

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

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

When you dimension an array, the system reserves space in internal memory for it. The system also sets up a table which it uses to locate each element in the array. The location of each element is designated by a unique combination of subscripts, one subscript for each dimension. For example,

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

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

---

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

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

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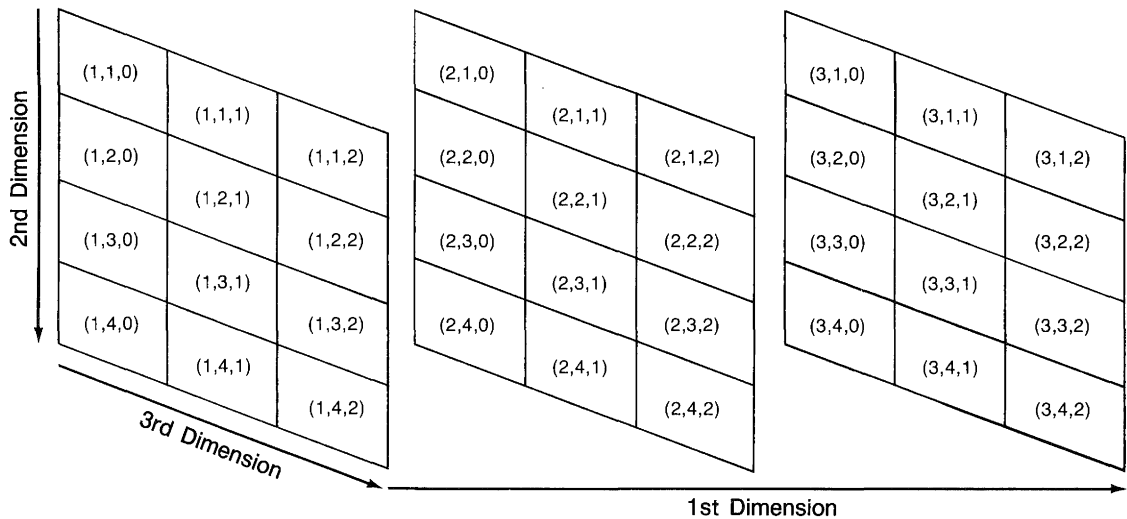
Some Examples of Arrays

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

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

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

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

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

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

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

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

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

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

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

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

Left array block:

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

Right array block:

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

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

Arrays are limited to six dimensions, and the subscript range for each dimension must lie between −32767 and 32767. (REDIM and ALLOCATE allow the subscript range to go down to −32768, but the total size of each dimension must be less than 32768 elements.) For the most part, we use only two-dimensional examples since they are easier to illustrate. However, the same principles apply to arrays of more than two dimensions as well.

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

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

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

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

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

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

then

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

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

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

dimensions all three arrays A, B, and C.

Problems With Implicit Dimensioning

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

Using Array Elements

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

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

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

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

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

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

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

Filling Arrays

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

- Assigning Every Element in an Array the Same Value.
- Using the READ Statement to Fill an Entire Array.
- Copying Arrays into Other Arrays.

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

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

assigns the value 10 to every element in array A, regardless of A's size. Note that the numeric expression on the right-hand side of the assignment must be enclosed in parentheses and that this expression may be INTEGER, REAL or COMPLEX. Let's look at an example of assigning a COMPLEX value to every element of a COMPLEX array:

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

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

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

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

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

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

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

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

```
MAT A=B
```

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

■ The two arrays must have the same rank (e.g., the same number of dimensions.)

■ The dimensioned size of the result array must be at least as large as the current size of the operand array.

If the system cannot redimension the result array to the proper size, it will return an error.

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

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

We can execute the statement

```
MAT A=B
```

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

```
MAT A=C
```

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

```
MAT A=C
```

you could not execute:

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

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

```
MAT C=A
```

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

Printing Arrays

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

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

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

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

Examples of Formatting Arrays for Display. This section provides two subprograms which have been given the name Printmat. The first subprogram is used to display two-dimensional arrays and the second subprogram is used to display three-dimensional arrays.

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

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

Assuming that the array you intend to display is a five-by-five two-dimensional array, your results should look similar to this:

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

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

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

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

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

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

Passing Entire Arrays

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

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

Copying Subarrays

Topics discussed in this section are as follows:

- Subarray specifier.
- Copying a subarray into an array.
- Rules for copying subarrays.

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

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

```
MAT Array55=Array33
```

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

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

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

```
Array_name(subarray_specifier)
```

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

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

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

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

Some examples of subarray specifiers are as follows:

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

2. ⟨1:2⟩ a given subscript range which represents the first two elements of a one-dimensional array.

3. ⟨*,-1:2⟩ a default range and subscript range which represents all of the elements in the first four columns of a two-dimensional array.

4. ⟨3,1:2⟩ a subscript expression and subscript range which represent the first two elements in the third row of a two-dimensional array.

5. ⟨1,*,*⟩ a subscript expression and two default ranges which represent a plane consisting of all the rows and columns of the first plane in the first-dimension.

6. ⟨1,1:2,*⟩ a subscript expression, subscript range and default range which represent the first two rows in the first plane of the first-dimension.

7. ⟨1,2,*⟩ two subscript expressions and a default range which represent the entire second row in the first plane of the first-dimension.

8. ⟨1:2,3:4⟩ two subscript ranges which represent elements located in the third and fourth columns of the first and second rows of a two-dimensional array. Copying an Array into a Subarray.

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

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

Suppose these arrays contain the following integer values:

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

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

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

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

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

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

- Subarray specifiers *must not* contain all subscript expressions (i.e. (1,2,3) is not allowed and it will produce a syntax error). This rule applies to all subscript specifiers.

- Subarray specifiers *must not* contain all asterisks (*) or default ranges (i.e. ⟨*,*,*⟩ is not allowed and it will produce a syntax error). This rule applies to all subscript specifiers.

- If two subarrays are given in a MAT statement, there *must be* the same number of ranges in each subarray specifier. For example,

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

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

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

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

- is legal; however,

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

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

Redimensioning Arrays

The system automatically redimensions an array during array assignment, if necessary. BASIC also allows you to explicitly redimension an array with the REDIM statement. As with automatic redimensioning, the following two rules apply to all REDIM statements:

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

Suppose A is the 3 × 3 array shown below.

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

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

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

The new array will look like the figure below:

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

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

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

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

We could then initialize all elements to 0:

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

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

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

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

results in:

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

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

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

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

Arrays and Arithmetic Operators

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

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

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

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

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

The result of the above addition is array B below:

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

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

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

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

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

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

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

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

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

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

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

would give the array:

$$\mathbf{D} = \begin{bmatrix} 5 & 7 & 9 \\ 11 & 13 & 15 \\ 17 & 19 & 21 \end{bmatrix}$$

The statement,

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

would give:

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

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

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

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

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

where the dimension statement is given as follows:

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

The element Op_array($-1,1$) contains the values:

```
2 4
```

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

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

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

The previous statement would produce the following complex array:

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

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

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

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

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

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

the function, SUM(A) would return 29.

Boolean Arrays

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

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

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

which results in the array:

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

---

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

Then you execute the statement,

```
PRINT SUM(F)
```

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

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

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

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

you could execute the statement:

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

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

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

String
Manipulation

It is often desirable to store non-numerical information in the computer. You can use any sequence of characters in a string. Quotation marks are used to delimit the beginning and ending of the string. The following are valid string assignments:

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

The left-hand side of the assignment (the variable name) is equated to the right-hand side of the assignment (the literal). String variable names are identical to numeric variable names with the exception of a dollar sign ($) appended to the end of the name. The length of a string is the number of characters in the string. In the previous example, the length of A$ is 8 since there are eight characters in the literal "COMPUTER". BASIC allows the dimensioned length of a string to range from 1 to 32,767 characters and the current length (number of characters in the string) to range from zero to the dimensioned length. A string of zero characters is often called a null string or an empty string. The default dimensioned length of a string is 18 characters. The DIM, COM, and ALLOCATE statements are used to define string lengths up to the maximum length of 32,767 characters. An error results whenever a string variable is assigned more characters than its dimensioned length. A string may contain any character. The only special case is when a quotation mark needs to be in a string. Two quotes, in succession, will embed a quote within a string:

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

Produces: The time is "NOW".

String Storage

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

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

The maximum length of any string must not exceed 32,767 characters. A string may also be dimensioned to a length less than the default length of 18 characters. The DIM statement reserves storage for strings:

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

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

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

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

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

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

String Arrays

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

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

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

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

Prints the 27th element in the array. Since each character in a string uses one byte of memory and each string in the array requires as many bytes as the length of the string, string arrays can quickly use a lot of memory. A program saved on a disc as an ASCII type file can be entered into a string array, manipulated, and written back out to disc.

Evaluating Expressions Containing Strings

This section covers the following topics:

■ Evaluation Hierarchy.

■ String Concatenation.

■ Relational Operations.

Evaluation Hierarchy. Evaluation of string expressions is simpler than evaluation of numerical expressions. The three allowed operations are extracting a substring, concatenation, and parenthesization.

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

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

When you run the program it will print the following:

```
WRIST WATCH WRISTWATCH
```

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

Relational Operations. Most of the relational operators used for numeric expression evaluation can also be used for the evaluation of strings. The following examples show some of the possible tests.

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

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

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

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

Note

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

Substrings

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

```
String$[4]
```

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

Single-Subscript Substrings. When a substring is specified with only one numerical expression enclosed with brackets, the expression is evaluated and rounded to an integer indicating the position of the first character of the substring within the string. The following examples use the variable A$ which has been assigned the literal "DICTIONARY".

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

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

Double-Subscript Substrings. A substring may have two subscripts, within brackets, to specify a range of characters. When a comma is used to separate the items within brackets, the first subscript marks the beginning position of the substring, while the second subscript is the ending position of the substring. For example:

```
"JABBERWOCKY"[4,6]
```

Specifies the substring "BER". When a semicolon is used in place of a comma, the first subscript again marks the beginning position of the substring, while the second subscript is now the length of the substring. For example:

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

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

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

An error results if the second subscript in a comma separated pair is greater than the current string length plus 1 **or** if the sum of the subscripts in a semicolon separated pair is greater than the current string length plus 1. Specifying the position just past the end of a string returns the null string.

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

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

When you run this program, it will print:

```
CONCATENATION
```

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

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

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

String-Related Functions

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

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

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

Prints: 7

Substring Position. You can determine the position of a substring within a string by the using the POS function. This function returns the value of the starting position of the substring, or zero if the entire substring was not found. For example:

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

Prints: 4

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

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

Prints: 123400

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

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

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

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

Prints: 65

Numeric-to-String Conversion. You can use the VAL$ function to convert the value of a numeric expression into a character string. The string contains the same characters (digits) that appear when the numeric variable is printed. For example:

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

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

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

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

Prints: abc

String Functions

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

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

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

Prints: snac kcanS

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

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

Prints: * ** ** ** ** ** ** ** ** *

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

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

Prints: *1.23*

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

Case Conversion. The case conversion functions, UPC$ and LWC$, return strings with all characters converted to the proper case. UPC$ converts all lowercase characters to their corresponding uppercase characters and LWC$ converts any uppercase characters to their corresponding lowercase characters. Roman Extension characters will be converted according to the current lexical order. See the LEXICAL ORDER IS statement later in this section for the case conversion listings.

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

MAT Functions and String Arrays

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

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

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

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

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

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

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

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

Running this program produces:

```
original order
Bread Milk Eggs Bacon Coffee

ascending order
Bacon Bread Coffee Eggs Milk

descending order
Milk Eggs Coffee Bread Bacon
```

Number-Base Conversion

Utility functions are available to simplify the calculations between different number bases. The two functions IVAL and DVAL convert a binary, octal, decimal, or hexadecimal string value into a decimal number. The IVAL$ and DVAL$ functions convert a decimal number into a binary, octal, decimal, or hexadecimal string value. Each function has two parameters: the string to be converted and the radix. The radix is limited to the values 2, 8, 10, or 16, and represents the numeric base of the string to be converted. The IVAL and IVAL$ functions are restricted to the range of INTEGER variables (−32,768 thru 32,767). The DVAL and DVAL$ functions allow "double length" integers and thus allow larger numbers to be converted (−2,147,483,648 thru 2,147,483,647). IVAL and IVAL$ operate on 16-bit values, while DVAL and DVAL$ operate on 32-bit values. The following statements show valid usage of these functions

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

Introduction to Lexical Order

The LEXICAL ORDER IS statement* lets you change the collating sequence (sorting order) of the character set.

Changing the lexical order will affect the results of all string relational operators and operations, including the MAT SORT and CASE statements. In addition to redefining the collating sequence, the case conversion functions, UPC$ and LWC$, are adjusted to reflect the current lexical order.

* Available with the LEX binary installed.

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

Predefined Lexical Order

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

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

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

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

* European Spanish keyboard.
† Latin Spanish keyboard.

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

User-Defined Functions and Subprograms

One of the most powerful constructs available in any language is the subprogram (a user-defined function is a special form of subprogram). A subprogram can do everything a main program can do except that it must be invoked or "called" before it is executed, whereas a main program is executed by pressing RUN or executing the RUN command.

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

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

Location

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

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

```
Initialize
Read_dvm
Sort_2_d_arry
Plot_data
```

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

The Difference Between a Function and a Subprogram

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

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

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

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

```
X=1/FNSinh(Y^4)
Angle=FNAtn2(Y,X)
```

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

REAL Precision Functions and String Functions

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

Calling and Executing a Subprogram

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

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

Communication

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

Parameter Lists. The formal parameter list is part of the subprogram's definition, just like the subprogram's name. The formal parameter list tells how many values may be passed to a subprogram, the types of those values (string, integer, real, array, I/O path name), and the names the subprogram will use to refer to those values. The subprogram has the power to demand that the calling context match the types declared in the formal parameter list—otherwise, an error results. It is perfectly legal for both the formal and pass parameter lists to be null, or nonexistent.

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

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

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

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

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

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

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

There are two ways for the calling context to send values to a subprogram: pass by value, and pass by reference. Using pass by value, the calling context supplies a value and nothing more. Using pass by reference, the calling context actually gives the subprogram access to the calling context's value area. The distinction is that a subprogram cannot alter the value of data in the calling context if the data is passed by value, while the subprogram can alter the value of data in the calling context if the data is passed by reference.

The subprogram has no control over whether its parameters are sent using pass by value or pass by reference. That is determined by the calling context's pass parameter list. In order for a parameter to be passed by reference, the pass parameter list (in the calling context) must use a variable for that parameter. In order for a parameter to be passed by value, the pass parameter list must use an expression for that parameter. Note that enclosing a variable in parentheses is sufficient to create an expression. Using pass by value, it is possible to pass an INTEGER expression to a REAL formal parameter (the INTEGER is converted to its REAL representation) without causing a type mismatch error. Likewise, it is possible to pass a REAL expression to an INTEGER formal parameter (the value of the expression is rounded to the nearest INTEGER) without causing a type mismatch error (an integer overflow error is generated if the expression is out of range for an INTEGER). Let's look at our previous example from the calling program:

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

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

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

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

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

Errflag is passed by reference.

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

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

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

A blank COM block might look like this:

```
10 OPTION BASE 1
20 COM Conditions(15),INTEGER,Cmin,Cmax,@Nuclear_pile,
   Pile_status$[20],Tolerance
```

A labeled COM might look like this:

```
30 COM /Valve/ Main(10,Subvalves910,15),@Valve_ctrl
```

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

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

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

- COM survives pre-run. In general, any numeric variable is set to 0, strings are set to the null string, and I/O path names are set to undefined when the program is run, or upon entering a subprogram. This is true of COM the first time the program is run, but after COM block variables are defined, they retain their values until:

  1. SCRATCH A or SCRATCH C is executed.

  2. A statement declaring a COM block is modified by the user.

  3. A new program is brought into memory using the GET or LOAD commands which doesn't match the declaration of a given COM block, or which doesn't declare a given COM block at all.

- COM blocks can be arbitrarily large. One limitation on parameter lists (both pass and formal parameter lists) is that they must fit into a single program line along with the line's number, possibly a label, the invocation or subprogram header, and possibly (in the case of a function) a string or numeric expression. Depending upon the situation, this can impose a restriction on the size of your parameter lists.

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

- COM blocks can be used for communicating between contexts that do not invoke each other.

- COM blocks can be used to communicate between subprograms that are not in memory simultaneously.

- COM blocks can be used to retain the value of "local" variables between subprogram calls.

- COM blocks allow subprograms to share data without the intervention of the main program.

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

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

1. COM blocks which match each other exactly between the two programs will be preserved intact. "Matching" requires that the COM blocks are named identically (except blank COM), and that corresponding blocks have exactly the same number of variables declared, and that the types and sizes of these variables match.

2. Any COM blocks existing in the old program which are not declared in the new program (the one being brought in with the LOAD or GET) are destroyed.

3. Any COM blocks which are named identically, but which do not match variables and types identically, are defined to match the definition of the new program. All values stored in that COM block under the old program are destroyed.

4. Any new COM blocks declared by the new program (including those mentioned above in #3) are initialized implicitly. Numeric variables and arrays are set to zero, strings are set to the null string, and I/O path names are set to undefined.

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

Consider the following COM block definition:

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

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

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

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

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

Context Switching

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

- The calling context has a DATA pointer which points to the next item in the current DATA block which will be used the next time a READ is executed. This pointer is saved away whenever a subprogram is called, and then the DATA pointer is reset to the first DATA statement in the new subprogram context.

- The RETURN stack for any GOSUBs in the current context is saved and set to the empty stack in the new context.

- The system priority of the current context is saved, and the called subprogram inherits this value. Any change to the system priority which takes place within the subprogram (or any of the subprograms which it calls in turn) is purely local, since the system priority is restored to its original value upon subprogram exit.

- Any event-initiated GOTO/GOSUB statements are disabled for the duration of the subprogram. If any of the specified events occur, this will be logged, but no action will be taken. Upon exiting the subprogram, these event-initiated conditions will be restored to active status, and if any of these events occurred while the subprogram was being executed, the proper branches will be taken.

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

- The current value of OPTION BASE is saved, and the value for the subprogram (0 or 1, explicitly declared or defaulted) is used.
- The current DEG or RAD mode for trigonometric operations and graphics rotations is stored away. The subprogram will inherit the current DEG or RAD setting, but if it gets changed within the subprogram, the original setting will be restored when the subprogram is exited.

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

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

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

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

Live Keyboard

Functions and subprograms can be called from live keyboard by the user. There are some restrictions:

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

Using Subprogram Libraries

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

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

Loading Subprograms One at a Time

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

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

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

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

Loading Several Subprograms at Once

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

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

and if the user selects Option 2,

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

and so forth.

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

Loading Subprograms Prior to Execution

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

Deleting Subprograms Programmatically

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

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

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

```
100 DELSUB Print_report TO END
```

You can combine these methods:

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

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

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

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

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

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

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

Editing Subprograms

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

1. STORE the program.
2. Delete all lines above the point where you want to insert your subprogram (refer to the DEL statement).
3. STORE the remaining segment of the program in a new file.
4. LOAD the original program stored in step 1.
5. Delete all lines below the point where you want to insert your subprogram.
6. Type in the new subprogram.
7. Do a LOADSUB ALL from the new file created in step 3.

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

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

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

Deleting Subprograms. It is not possible to delete either DEF FN or SUB statements with DEL LINE unless you first delete all the other lines in the subprogram. This includes any comments after the SUBEND or FNEND. Another way to delete DEF FN and SUB statements is to delete the entire subprogram, up to, but not including, the next SUB or DEF FN line (if any). This can be done either with the DEL command, or with the DELSUB command.

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

1. SAVE everything in your program after the SUB or DEF FN statement you want to delete.

2. Delete everything in your program from the unwanted SUB statement to the end.

3. GET the program segment you saved in step 1 back into memory, taking care to number the segment in such a way as not to overlay the part of the program already in memory.

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

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

SUBEND and FNEND

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

Recursion

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

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

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

References

1. Wirth, Niklaus, "Program Development by Stepwise Refinement", *Communications of the ACM*, April 1971, Vol. 14, No. 4, pp. 221-227

2. Yourdan, Edward, *Techniques of Program Structure and Design*, (Prentice-Hall, Englewood Cliffs, NJ, 1975)

3. Dahl, Dijkstra, & Hoare, *Structured Programming* (Academic Press, New York, 1972)

Data Storage and Retrieval

This section describes some useful techniques for storing and retrieving data. First we describe how to store and retrieve data that is part of the BASIC program. With this method, DATA statements specify data to be stored in the memory area used by BASIC programs; thus, the data is always kept with the program, even when the program is stored in a mass storage file. The data items can be retrieved by using READ statements to assign the values to variables. This is a particularly effective technique for small amounts of data that you want to maintain in a program file.

For larger amounts of data, mass storage files are more appropriate. Files provide means of storing data on mass storage devices. The two types of data files, ASCII and BDAT, are described in this section. A number of different techniques for accessing data in BDAT files are described in detail.

The BASIC system can use a number of different mass storage devices, including internal disc drives, external disc drives, "memory volumes" and SRM systems. This section gives guidelines for accessing many kinds of devices.

Storing Data in Programs

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

Storing Data in Variables

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

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

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

Data Input by the User

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

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

200 DISP "Enter the value of X,Y, and Z."

300 LINPUT Response$
```

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

Using DATA and READ statements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The date is November 26, 1981.
```

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

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

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

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

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

The Structure of Data Files

There are two file types that you can use to store data: BDAT and ASCII. BDAT files have several advantages: they allow more flexibility in data formats and access methods, allow faster transfer rates, and are generally more space-efficient than ASCII data files. They can be randomly or serially accessed, and they allow data to be stored in either ASCII format, internal format, or a specialized format (defined by the user with IMAGE statements).

ASCII files allow *only* serial access and *only* ASCII format. They have these advantages: the files are compatible with other HP computers that support this file type, the format provides very compact storage for string data, and there is no chance of reading the contents into the wrong data type (a problem with BDAT files). The full name of ASCII files is "LIF ASCII." LIF stands for Logical Interchange Format, a directory and data storage format that is used by many HP computer divisions. Understanding the characteristics of each file type will help you choose the best one for your specific application.

BDAT Files. BDAT files are designed to be storage-space efficient, have high data-transfer rates, and allow *both* random and serial access. Random access means that you can directly read from and write to any record within the file, while serial access only permits you to access the file from the beginning. Serial access can waste a lot of time if you're trying to access data at the end of a file. On the other hand, if you want to access the entire file sequentially, you are better off using serial access than random access. BDAT files can be accessed both randomly and serially, while ASCII files can only be accessed serially.

BDAT files allow you to store and retrieve data using internal format, ASCII format, or user-defined formats. With internal format, items are represented with the same format the system uses to store data in internal computer memory*. With ASCII format, items are represented by ASCII characters. User-defined formats are implemented with programs that employ OUTPUT and ENTER statements that reference IMAGE specifiers.

---

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

In most applications, you will use internal format for BDAT files. Unless we specify otherwise, you can assume that when we talk about retrieving and storing data in BDAT files, we are also talking about internal format.

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

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

INTEGER 2 bytes

REAL 8 bytes

String 1 byte per character (plus 1 pad byte if the string length is an odd number), plus a 4-byte length header

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

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

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

STRING data are stored in BDAT files in their internal format (plus two additional, leading bytes of length header, which are always 0 for Series 200/300 computers). Every character in a string is represented by one byte which contains the character's ASCII code. The four-byte length header contains a value that specifies the length of the string. If the length of the string is odd, a pad character is appended to the string to get an even number of characters; however, the length header does not include this pad character.

The string "A" would be stored:

00000000 00000000 00000000 00000001 01000001 00100000

      Length = 0001 (binary)   ASCII 65 ASCII 32

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

When using the ASCII data format for BDAT files, all data items are represented with ASCII characters. With user-defined formats, the image specifiers referenced by the OUTPUT or ENTER statement are used to determine the data representation. Using both of these formats with BDAT files produce results identical to using them with devices.

ASCII Files. You have already been introduced to ASCII files as a way to SAVE programs. ASCII files can also be used to store data. In an ASCII file, every data item, whether string or numeric, is represented by ASCII characters; one byte represents one ASCII character. Each data item is preceded by a two-byte length header which indicates how many ASCII characters are in the item. However, there is no "type" field for each item; data items contain no indication (in the file) as to whether the item was stored as string or numeric data.

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

```
┌───┬───┬───┬───┬─────┬───┬──────╮
│ 0 │ 3 │   │ 1 │ 2 │(pad)│   │ ••• │
└───┴───┴───┴───┴─────┴───┴──────╯
  ╰──┬──╯     ╰────┬────╯
  LENGTH        ASCII
  HEADER =      CODES
  BINARY 3
```
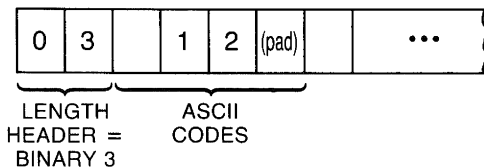
Similarly, reading numeric data from an ASCII file can be a complex and relatively slow operation. The numeric characters in an item must be entered and evaluated individually by the system's "number builder" routine, which derives the number's internal representation. (Keep in mind that this routine is called automatically when data are entered into a numeric variable.) For example, suppose that the following item is stored in an ASCII file:

```
┌───┬────┬───┬───┬───┬───┬───┬───┬───┬───┬───┬──────╮
│ 0 │ 10 │ A │ B │ C │ = │   │ 1 │ 2 │ 3 │ X │ Y │   │ ••• │
└───┴────┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴──────╯
  ╰───┬───╯   ╰──────────────┬──────────────╯
   LENGTH                 ASCII
   HEADER =               CODES
   BINARY 10
```
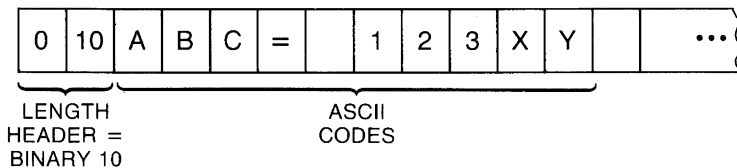
Although it may seem obvious that this is not a numeric data item, the system has no way of knowing this since there is no type-field stored with the item. Therefore, if you attempt to enter this item into a numeric variable, the system uses the number-builder routine to strip away all non-numeric characters and spaces and assign the value 123 to the numeric variable. When you add to this the intricacies of real numbers and exponential notation, the situation becomes more complex.

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

Mass Storage Techniques

This section presents BASIC programming techniques useful for accessing mass storage devices and files. The first part gives a brief introduction to the steps you might take to store data in a file. Subsequent parts describe further details of these steps. If you feel that you need additional background information while reading this material, refer to the preceding tutorial section.

Overview of Mass Storage Access

Storing data in files requires a few simple steps. The following program segment (found in file CRBDAT in your Manual Examples disc) shows a simple example of creating a BDAT file on the "A" drive, and writing data to it.

```
10   DIM Array1(5,4),Array2(5,4)
20   MAT Array1=5 !Fill Array1 with 5.
30   MASS STORAGE IS ":CS80,1500,0" ! MSI is drive A.
40   CREATE BDAT "FILE_1",10 ! 10 (256-byte) records.
50   ASSIGN @Path_1 to "FILE_1" ! Open an I/O path to the file.
60   OUTPUT @Path_1;Array1(*) ! Send an array of numeric values.
70   ASSIGN @Path_1 TO * ! Close the path
80   ASSIGN @F_1 to "FILE_1:CS80,1500,0" ! Another path to the file.
90   ENTER @F_1;Array2(*)
100  ASSIGN @Path_1 TO * ! Close the path
110  END
```

Line 30 specifies the "system mass storage device," or the "default" device which is to to be used whenever a mass storage device is not explicitly specified during subsequent mass storage operations. The term *mass storage unit specifier (msus)* describes the string expression used to uniquely identify which device is to be the mass storage. In this case, ":CS80,1500,0" is the msus.

In order to store data in mass storage, a data file must be created (or already exist) on the mass storage media. In this case, line 40 creates a BDAT file for data storage; the file created contains 10 defined records of 256 bytes each.

The term "file specifier" describes the string expression used to uniquely identify the file. In this example, the file specifier is simply File_1, which is the file's name. If the file is to be created (or already exists) on a mass storage device *other than the system mass storage*, the appropriate msus must be appended to the file name.

Then, in order to store data in (or retrieve data from) the file, you must assign an I/O path name to the file. Line 50 shows an example of assigning an I/O path name to the file (also called opening an I/O path to the file). Line 60 shows an array of numeric data being sent to the file through the I/O path.

The I/O path is closed after all data have been sent to the file. Closing the I/O path is optional if another I/O path name is assigned to the file later in the program. All I/O path names are automatically closed by the system at the end of the program. Closing an I/O path to a file updates the file pointers.

If this array of data is to be retrieved from the file, another ASSIGN statement is executed (line 110). Notice that a different I/O path name has been used; this is an arbitrary choice of names. Opening this I/O path name to the file sets the file pointer to the beginning of the file. (Re-opening the I/O path name @File_1 would have also reset the file pointer.)

Notice also that the msus is included with the file name. This shows that the mass storage device, does not have to be the current system mass storage in order to be accessed. The subsequent ENTER statement reads the data into another numeric array (which must be of the *same* data type when a BDAT file is used in this manner).

Non-Disc Mass Storage

Although mass storage is traditionally implemented using a magnetic surface such as a disc or drum, the protocols of file management can be applied to any device which stores data, such as RAM Memory Volumes. Areas of the computer's RAM can be treated as though they were mass storage devices. Obviously, a RAM volume is volatile. However, it can be accessed faster than any other mass storage device.

RAM Volumes. Areas of the computers RAM may be treated as mass storage devices. These "memory volumes" or "RAM volumes" are volatile (all information is lost when the power goes off), but high speed. A typical use for RAM volumes is to copy a disc volume into memory, perform all necessary manipulations using the RAM volume, then copy the new information back to disc. Obviously, there are only certain applications which would benefit from this technique.

All mass storage operations work with RAM volumes.

RAM volumes are created by the INITIALIZE statement. A special form of this statement is used, with a unit size parameter in the position normally occupied by the interleave factor. The device type is always MEMORY, and the device selector is always 0. Unit numbers 0 thru 31 may be used. Here are some examples.

```
INITIALIZE ":MEMORY,0,7" ,220
```

This creates a RAM volume that is 220 sectors long and is given unit number 7. Note that the unit size parameter is in 256-byte sectors, just like LIF file sizes.

If the unit size parameter is omitted, the result is a RAM volume that is the same size as a 5.25-inch or 3.5-inch disc. This is 1056 sectors, or 270,336 bytes. The default size RAM volume provides only 80 directory entries, while the discs may contain up to 112 directory entries. If a disc is copied into the RAM volume, the entire directory will be copied.

The unit size of a RAM volume must be at least 4 sectors and can be as large as available memory permits. Two sectors are taken for system use, and about 1 sector of directory is created for each 100 sectors of unit size.

No RAM volumes exist at power-up or after a SCRATCH A. It is recommended that all binaries be loaded before RAM volumes are initialized. If a binary is loaded after a RAM volume is initialized, the memory used for the RAM volume cannot be recovered until the computer is turned off and back on again.

A RAM volume can be re-initialized to the same or different size. If the size is different, memory space may be lost until the next SCRATCH A.

After they are created, RAM volumes are accessed by using their unit number in a MEMORY media specifier. The following examples show typical mass storage unit specifiers for a RAM volume with unit number 7.

```
MASS STORAGE IS ":MEMORY,0,7"
or
ASSIGN @Ram TO "TEMP:MEMORY,0,7"
```

Accessing Files

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

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

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

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

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

```
ASSIGN @Path2 TO "Example:CS80,1500,0"
```

opens an I/O path to the file Example. You must include the protect code if the file has one.

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

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

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

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

```
ASSIGN @Path2 TO *
```

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

Reading and Writing BDAT Files

There are many alternatives for storing and retrieving data when using BDAT files. You can choose internal, ASCII, or user-defined formats, and serial or random access.

System Sector

On the disc, every BDAT file is preceded by a system sector that contains an End-Of-File pointer and the number of defined records in the file. All data is placed in succeeding sectors. You cannot directly access the system sector. However, as you shall see later, it is possible to indirectly change the value of an EOF pointer.
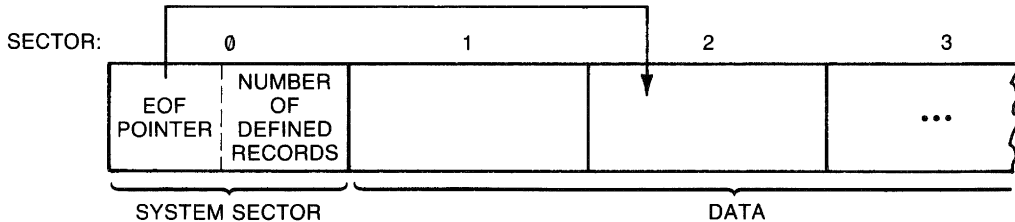
EOF Pointer: • number of sectors from beginning of file
(32-bit binary number)
• number of bytes from beginning of sector
(32-bit binary number)
Number of defined records: See description below
(32-bit binary number)

Defined Records

To access a BDAT file randomly, you specify a particular defined record. Records are the smallest units in a file directly addressable by a random OUTPUT or ENTER. They can be anywhere from 1 through 65,534 bytes long. Both the length of the file and the length of the defined records in it are specified when you create the file. For example, the statement:

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

would create a file called Example with 7 defined records, each record being 128 bytes long. If you don't specify a record length in the CREATE BDAT statement, the system will set each record to the default length of 256 bytes.

Both the record length and the number of records are rounded to the nearest integer. Further, the record length is rounded up to the nearest even integer. For example, the statement:

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

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

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

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

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

Choosing a Record Length

The most important consideration in selecting of a proper record length is the type of data being stored and the way you want to retrieve it. For optimum performance, the record size should be an even multiple of the size of the data elements stored in the record, 2 bytes for integers and 8 bytes for real numbers.

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

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

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

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

```
100 CREATE BDAT "Names",5,18 ! Create a file.
110 ASSIGN @File TO "Names" ! Open an I/O path
120 OUTPUT @File,1;"John Smith" ! Write names to
130 OUTPUT @File,2;"Steve Anderson" ! Successive records
140 OUTPUT @File,3;"Mary Martin" ! In file
150 OUTPUT @File,4;"Bob Jones"
160 OUTPUT @File,5;"Beth Robinson"
```

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

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

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

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

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

Writing Data

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

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

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

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

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

Sequential (Serial) OUTPUT

Data is written sequentially (serially) to BDAT files whenever you do not specify a record number in an OUTPUT statement. When data is written serially, each data item is stored immediately after the previous item without any type of separator. Sector and record boundaries are ignored. Data items are written to the file one by one, starting at the current position of the file pointer. As each item is written, the file pointer is moved to the next byte. After all of the data items have been OUTPUT, the file pointer points to the first byte following the last byte just written.

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

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

Random OUTPUT

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

If you attempt to write more data to a record than the record will hold, the system will return an End-Of-Record (EOR) condition. An EOF condition will result if you try to write data more than one record past the EOF position. EOR conditions are treated by the system just like EOF conditions, except that they return Error 60 instead of 59 if they are not trapped by ON END. Data already written to the file before an EOR condition arises will remain intact.

Reading Data From BDAT Files

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

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

When reading data into a string variable, it is important to remember that the system will interpret the first four bytes after the file pointer as a length header. It will then try to ENTER as many characters as the length header indicates. If the string has been padded by the system to make its length even, the pad character is not read into the variable.

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

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

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

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

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

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

5 squared equals 25.
```

Note that we re-ASSIGNed the I/O path in line 70. This was done to re-position the file pointer to the beginning of the file. If we had omitted this statement, the ENTER would have produced an EOF condition. Note also that the OUTPUT statement includes END, which specifies that the EOF pointer is to be moved to match the file pointer at statement completion. In this case, the END is redundant.

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

In the following example (found in file OUTPUT1 on your Manual Examples disc), data is randomly OUTPUT to 10 successive records, and then ENTERed into an array in reverse order.

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


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

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

Executing a random ENTER without a variable list has the effect of moving the file pointer to the beginning of the specified record. This is useful if you want to serially access some data in the middle of a file.

You can define records to be just one byte long. In this case, it doesn't make sense to read or write one record at a time, since even the shortest data type requires two bytes to store a number.

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

General Mass Storage Operation

This section describes several different types of operations on mass storage files.

- Trapping EOR and EOF conditions while reading and writing data files
- Protecting files
- Copying files
- Purging files
- Accessing directories programmatically

Trapping EOF and EOR Conditions

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

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

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

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

Protecting Files*

Note

File protection does not prevent MS-DOS read/write of files, or copying files with the HPWUTIL utility.

Protect codes are two-character strings that can be assigned to any BDAT, BIN or PROG type file with the PROTECT statement. Protect codes are not unbreakable; they are only intended to prevent accidentally writing in files and directories.

For instance, the following statement assigns the protect code "AA" to the file named "FILE1."

```
PROTECT "FILE1","AA"
```

---

* This type of protect code applies only to non-SRM LIF discs. For a description of SRM password protection, refer to Chapter 6, "Using SRM."

File specifiers in mass storage statements that write to a file or directory must include the protect code, if the file has one. Mass storage statements that read a file or directory (CAT, LOAD, LOAD BIN, LOADSUB ALL FROM, GET and COPY) do not require the protect code. A protect code is specified by placing it in brackets right after the file name. To assign an I/O path name to the file named "FILE1," you would now have to include the protect code.

```
ASSIGN @Path1 TO "FILE1<AA>"
```

If you assign a protect code longer than two characters, the system will ignore everything after the second (non-bland) character. For example, the protect codes LONGPASS, LOLLYPOP, and LOST all result in the same protect code: LO. This rule holds both for PROTECTing a file and for specifying the protect code in a file specifier. For example:

```
PROTECT "FILE1","Protect1"
```

assigns the protect code "Pr" to FILE1. To rename the file, you could write:

```
RENAME "FILE1<Prattle>" TO "FILE2"
```

"Prattle" is an acceptable protect code, since it starts with "Pr." Note that we do not include a protect code in the new file name. If you do, the system ignores it since the old protect code is passed to the new file name. FILE2 still has the protect code "Pr". To rename the file again, we might write:

```
RENAME "FILE2<Pr>" TO "FILE3"
```

Renaming a file has the effect of changing the file name in the directory and leaving everything else intact.

In addition to using the PROTECT statement, you can also assign a protect code to a BDAT file when you create it. For example:

```
CREATE BDAT "Example<Xxx>",10
```

creates a 10-record BDAT file called "Example" and gives it a protect code of "xx". You can also do this to PROG files with the STORE and STORE BIN statements. However, since ASCII files cannot be protected, a protect code cannot be included in any CREATE ASCII, SAVE, or RE-SAVE statement.

To change a protect code, simply execute a new PROTECT statement. To change the protect code of "Example" to "yy," execute:

```
PROTECT "Example<xx>","yy"
```

Note that you must include the current protect code in the file specifier.

To completely remove a protect code from a file, PROTECT the file with a code consisting of two blanks. For example, to remove the protect code from file "Example," execute:

```
PROTECT "Example<yy>","  "
```

When specifying a file that does not have a protect code, you can either ignore the code entirely or include a code of two spaces:

```
PURGE "Example"
```

or

```
PURGE "Example<  >"
```

Copying Files

The COPY statement allows you to duplicate individual files. Any type of file may be copied.

COPY of a file duplicates the existing file and places the new file name in the directory. A new file can be created either on the same disc or on another disc. If you copy a file to the same disc, the new file name must be different from the existing file name. If the file is of BDAT, BIN or PROG type, you can also assign a protect code to the new file. If there is not enough room on the disc for the file to be copied, the system cancels the statement and returns an error.

Caution

Copying entire directories or volumes to or from an internal drive should be accomplished with the HPWUTIL program. Refer to appendix C for information on this utility.

Examples. The following statement copies "File1" from the current system mass storage device to a new file called "File2" on the same mass storage.

```
COPY "File1" TO "File2"
```

The following statement copies "File1" from the current system mass storage to the drive at interface select code 15, primary address 0, unit number 0. Note that both files can be named "FILE1 if they are on different volumes.

```
COPY "File1" TO "File1:CS80,1500,0"
```

The following statement copies a file from a disc drive to the current system mass storage device. The new file "DATA" is given the protect code"xx."

```
COPY "File1:CS80,1500,0" TO "DATA<xx>"
```

Purging Files

You can purge a file by using the PURGE statement. Purging a file deletes the directory entry for the file and releases the reserved space in the data area. Purging a file, therefore, creates two "gaps" on the disc: one in the data area and one in the directory. When you create a file, the system looks at all the gaps in the data area to see if the newly created file will fit in any of them.

Accessing Directories

Disc structure and mass storage directories were briefly described earlier in this section. As you may recall, a directory is merely an index to the files on a mass storage media. The BASIC language has several features that allow you to obtain information from the directories of mass storage media. This section presents several techniques that will help you access this information.

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

```
CAT
```

Including a media specifier directs the system to get a catalog of the specified mass storage. For instance, executing the following statement returns a catalog of the directory of the "A" drive:

```
CAT ":CS80,1500,0"
```

Both of the preceding statements sent the catalog listings to the current system printer (the one specified in the last PRINTER IS statement; the default system printing device is the CRT).

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

```
CAT TO #26
CAT TO #External_prtr
CAT TO #Device_selector
```

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

Cataloging Selected Files. The directory entry of file(s) that begin with certain character(s) can be obtained by using the secondary keyword SELECT. Suppose that you want to catalog only files beginning with the letters "Prog". The folowing examples show how this may be accomplished. Notice that this is *not* the same opertion as getting a catalog of a PROG file.

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

The directory entries of the three files beginning with the letters "Prog" are sent to the PRINTER IS device. In the second CAT statement above, the variable Files_and_headr is filled with the number of selected files found on the current default mass storage device (plus the 5 header lines). (Keep in mind that the variable Files_and_headr must be currently defined in the current program context.)

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

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

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

```
10 CAT;COUNT Files_and_headr
20 END

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

The first CAT operation returns a count of all files in the directory (plus 5 header lines), since not including SELECT defaults to "select all files". The second operation returns a count of the specifically selected files (plus 5).

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

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

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

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

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

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

Using a Printer

Sooner or later it needs to be printed. A wide range of printers, supported by BASIC, can be connected to your computer. This section covers the statements commonly used to communicate with external printers. The following is a list of some of the printers that work with most popular personal computers:

- HP 2225 Thinkjet Printer
- HP 2601 Daisy-Wheel Printer
- HP 2631 Dot Matrix Printer
- HP 2671 Thermal Printer
- HP 2686 Laser Printer
- HP 82906 Dot Matrix Printer

Fundamentals

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

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

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

Device Selectors

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

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

```
PRINTER IS 1
```

This statement defines the screen of the CRT to be the system printer. Until changed, the output of PRINT statements will appear on the screen of the CRT.

Primary Addresses

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

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

```
PRINTER IS 701
```

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

Using Device Selectors

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

PRINTER IS 1 Specifies the internal CRT (default).

PRINTER IS 701 Specifies an HP-IB printer with interface select code 7 and primary address 01.

PRINTER IS 26 Specifies a printer with interface select code 26. This will access a standard MS-DOS printer at LPT1.

PRINTER IS 22 Specifies a printer connected to interface select code 22.

CAT TO #701 Prints a disc directory at 701.

PRINTALL IS 707 Logs information on a printer whose select code is 7 and whose primary address is 07 (binary 00111).

LIST #701 Lists the program in memory to a printer connected to the internal HP-IB interface at primary address 01.

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

```
PRINTER IS Hal

CAT TO #Dog
```

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

Mnemonic	Value
PRT	701
KBD	2
CRT	1

For example, the following statements perform the same action:

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

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

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

Using the External Printer

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

Control Characters

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

```
PRINTER IS 26
PRINT CHR$(12)
```

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

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

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

Formatted Printing

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

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

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

Produces:

```
12345678901234567890123456789012345678
 1.1      -22.2     300000    5.1E+8
```

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

```
A;B;C;D,E
123456789012345678901234567890123
 1.1 -22.2  300000  5.1E+8
```

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

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

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

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

Produces:

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

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

```
PRINT TAB(25);-1.414

123456789012345678901234567890123
                        -1.414
```

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

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

Using Images

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

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

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

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

```
3.142
```

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

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

3.1415926536
```

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

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

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

Both PRINT statements use the image in line 100.

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

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

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

Statement	Output
PRINT USING "K";33,666	33.666
PRINT USING "DD.DDD";33.666	33.666
PRINT USING "DDD.DD";33.666	33.67
PRINT USING "ZZZ.DD";33.666	033.67
PRINT USING "ZZZ";.444	000
PRINT USING "ZZZ";.555	001
PRINT USING "SD.3DE";6.023E+23	+6.123E+23
PRINT USING "S3D.3DE";6.023E+23	+602.300E+21
PRINT USING "S5D.3de";6.023E+23	+60230.000E+19
PRINT USING "H";3121.55	3121.55
PRINT USING "DDRDD";19.95	19.95
PRINT USING "***";.555	**1

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

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

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

```
  PRINT USING "5D.DD";3.98,5.95,27.50,139.95
12345678901234567890123456789012 3
    3.98    5.95   27.50  139.95
```

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

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

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

The following examples show various ways to use string specifiers.

```
PRINT USING "5X,10A,2X,10A";"Tom","Smith"
12345678901234567890123456789
     Tom           Smith

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

12345678901234567890123456789
     John   Smith

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

12345678901234567890123456789
PART NUMBER    90001234
```

Additional Image Specifiers. The following image specifiers serve a special purpose.

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

For example:

```
PRINT USING "@,#" outputs a formfeed.
PRINT USING "D,X,3A,""OR NOT"",X,B,X,B,B";2,"BE",50,66,69
```

Special Considerations

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

Sending text to a non-existent printer will cause the computer to wait indefinitely for the printer to respond. ON TIMEOUT may be used within a program to test for the printer. To clear the error press CLEAR I/O check the interface cable, and switch settings then try again.

The Real-Time Clock

Most personal computers have a real-time clock that you can set and read to monitor the time of day and date. The clock keeps time even when the power is removed from the computer. This section describes using the clock and related functions and statements.

Many of the statements described in this section require the CLOCK binary. Refer to the *BASIC Language Reference* manual for specific requirements of each statement.

Clock Range and Accuracy

The range of the clock is March 1, 1900 through August 4, 2079. The clock maintains time to within ±2.5 seconds per day.

Initial Clock Value

When you boot the HP BASIC system, the HP BASIC clock in most personal computers is set to one of three values:

- The HP BASIC clock time is set to the value of the real-time clock. If there is no real-time clock, the HP BASIC clock is set to 12:00:00 (midnight), March 1, 1900.
- With computers on the Shared Resource Management (SRM) system that *don't* have a real-time clock, the clock value is taken from the SRM system. (This occurs only when the SRM and DCOMM binaries are loaded.)

Reading the Clock

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

```
PRINT TIMEDATE
```

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

Determining Date and Time of Day

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

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

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

The TIME$ function returns the time of day.

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

Setting the Clock

The SET TIMEDATE statement is used to set the clock.

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

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

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

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

Setting the Time

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

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

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

Is equivalent to:

```
SET TIME 55810
```

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

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

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

Prints: 15:30:16

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

Setting the Date

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

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

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

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

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

Both of these statements are equivalent to the following statement.

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

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

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

Prints: 1 Jun 1984

Day of the Week

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

Branching on Clock Events

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

■ ON TIME enables a branch to be taken when the clock reaches a specified time of day.

■ ON DELAY enables a branch to be taken after a specified number of seconds has elapsed.

■ ON CYCLE enables a recurring branch to be taken with each passage of a specified number of seconds.

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

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

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

Cycles and Delays

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

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

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

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

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

For example:

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

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

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

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

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

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

Priority Restrictions

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

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

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

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

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

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

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

Branching Restrictions

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

Error Handling

Most programs are subject to errors happening at run time. There are three courses of action to take with respect to errors:

1. Try to prevent the error from happening in the first place.

2. Once an error occurs, try to recover from it and continue execution.

3. Do nothing—let the program "roll over and die" if an error happens.

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement, and the nature of most personal computers is such that this is often a feasible choice. Upon encountering a run-time error, the computer will pause program execution and display a message giving the error number and the line in which the error happened, and the

programmer can then examine the program in light of this information and fix things up. The key word here is "programmer." If the person running the program is also the person who wrote the program, this approach works fine. If the person running the program did not write it, or worse yet, does not know how to program, some attempt should be made to prevent errors from happening in the first place, or to recover from errors and continue running.

Anticipating Operator Errors

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

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

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

Error Trapping

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

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

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

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

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

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

The DISABLE statement has no effect on ON ERROR branching.

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

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

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

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

```
1140 DISP ERRL(710)

910 IF ERRL(Compute) THEN Fix_compute
```

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

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

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

By using the ON ERROR GOTO statement, the same error service routine can be used to service all the error conditions in a given context. By testing both the ERRN (what went wrong) and the ERRL (where it went wrong) functions, proper recovery procedures can be taken. Load and run file ERRECOVER on the examples disc for a detailed example.

ON ERROR CALL. ON ERROR CALL is global, meaning once it is activated, the specified subprogram will be called immediately whenever an error is encountered, regardless of the current context. System priority is set to level 16 inside the subprogram, and remains that way until the SUBEXIT is executed, at which time the system priority will be restored to the value it was when the error happened.

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

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

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

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

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

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

Program Debugging

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

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

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

Using Live Keyboard

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

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

```
10 FOR I=1 TO 1.E+5
20 NEXT I
30 END

CAT
2+2
SQR(6^2+17.2^2)
PRINT "THE QUICK BROWN FOX"
TIMEDATE
```

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

```
I=99999
```

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

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

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

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

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

```
ASSIGN @Dvm TO *
```

from the keyboard, but it is not legal to perform

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

from the keyboard.

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

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

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

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

Stepping

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

There are three ways to use STEP*:

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

2. If the program is in an INPUT or LINPUT statement, pressing STEP* is sufficient to terminate the operation. Any data entered from the keyboard will be entered into the correct variables, just as though CONTINUE or (Enter) had been pressed, but program execution will be PAUSEd, and the statement immediately following the INPUT or LINPUT will appear in the system message line.

3. If the program is in a PAUSEd state, pressing STEP* will cause the next line to be executed. The program counter will not be reset, nor will a prerun be performed. Again, the next line to be executed will appear in the system message line after the last one has been completed. A paused state is indicated by a dash in the run light in the lower right-hand corner of the screen.

---

* Refer to appendix E or the Key Function and Switch Configuration Guide to find the keystrokes for STEP.

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

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

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

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

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

Press PRINT ALL (refer to appendix E or the Key Function and Switch Configuration Guide) to turn on PRINT ALL. Load and run the following example (found in file TRACEALL on your Manual Examples disc) to see how TRACE ALL works:

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

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

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

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

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

PRINTALL IS

The PRINTALL IS command is useful for switching the tracing messages between the CRT and a hardcopy printer. (Again, to get any record at all of the trace messages, PRINT ALL must be on.) To cause the trace messages to be logged on the CRT, execute PRINTALL IS CRT. (The CRT is the default PRINTALL IS device that the system assumes when it wakes up.) To cause the messages to be logged on a printer, merely change the select code to the appropriate value (PRINTALL IS 26).

TRACE PAUSE

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

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

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

Try the following ways of continuing execution:

- Press STEP.
- Press CONTINUE.
- Execute CONT 110 (Enter).

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

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

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

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

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

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

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

4 Graphics Techniques

Contents

4

Graphics Techniques

Getting Started

Graphics is a good means of presenting information. This chapter takes you step-by-step through the graphics design process in order to give you a basic background in graphics programming on your computer. If you're an expert graphics programmer, you may want to skip directly to appendix A to familiarize yourself with the graphics statements. If you're not familiar with graphics concepts, read this chapter carefully and completely.

You should try the examples on your computer as you go through this chapter. If you have not loaded the GRAPH and GRAPHX binaries, do so now. Refer to chapter 2 for information on loading binaries.

Graphics Information

Before you create graphics, here is some background information about your graphics system.

The CRT Display

You will use the internal CRT as a plotter as you progress through this chapter. Later, other kinds of plotters are explained. This is because it is easier to develop and edit graphics programs using the CRT. With minor changes, the image can be reproduced on an external plotter.

The Alpha and Graphics Displays. The CRT has two separate displays, an alpha display and a graphics display, which can be output either individually or combined. The alpha display outputs alphanumeric characters such as error messages or commands, while the graphics display, obviously, outputs graphics.

The alpha display is controlled with:

```
ALPHA ON
```

and

```
ALPHA OFF
```

The graphics display is controlled with:

```
GRAPHICS ON
```

and

```
GRAPHICS OFF
```

When you turn the computer on, the alpha display is on and the graphics display is off. During execution of a graphics program, the alpha display may be turned off, but execution of a command or sending output to the alpha display turns the alpha display on and leaves it on. The graphics display can only be controlled with explicit GRAPHICS ON and GRAPHICS OFF statements.

Resolution. A notable difference between CRTs and other plotters is the resolution of the display. You probably know that the CRT consists of an array of pixels or picture elements. The resolution of graphics is directly dependent upon the number of pixels per unit area.

The Current Position

When dealing with graphics output, this text uses the concept of a current position. It is the point relative to which graphics are currently output. Usually you can think of this as the pen's current location or the location at which graphics can be currently output.

The current position is not always where the physical pen is located. For example, if you instruct the pen to move to a point outside the edge of the plotter, the physical pen only moves to the edge, but the current position is updated to the point specified.

Although the current position is referred to when discussing where graphics are output, the concept of a pen is still important. Knowledge of what the pen type is and whether it is "up" or "down" is needed. Thus the pen, on a CRT, is defined as the effect which gives the appearance of an invisible pen creating lines on the display. On a paper plotter, it is the control arm which holds the ink pens.

Preparing to Output

To bring the system to a known starting point, execute the command:

```
GINIT
```

This initializes the graphics in the system by resetting all the attributes, viewing operations, plotters and other system variables. GINIT should always be executed before starting any graphics programming.

Clearing the Displays

Once GINIT is executed, you want to make it easy to see the graphics display. One problem encountered is that data in the alpha display covers the graphics display. This can be removed. To save the information currently in the alpha display, simply execute:

```
ALPHA OFF
```

To delete the data in the alpha display, press CLEAR SCREEN. A formfeed, CHR$(12) also clears the alpha display. The cursor is the only thing left on the alpha display. CLEAR SCREEN does not affect the graphics display, so you don't have to worry about accidentally deleting your work.

To clear the graphics display, execute:

```
GCLEAR
```

Any graphics in the output area of the display are lost, so be sure that's what you want to do. This does not affect the alpha display.

If you execute GINIT, any subsequent output statement also causes a clearing of the graphics display on the internal CRT.

The XY Plane

Graphics primitives are output on an imaginary plane known as the XY plane; X is a horizontal axis and Y is a vertical axis on this plane. Any two-dimensional images which you create are assigned a position on this plane using XY coordinates. Obviously, this plane cannot be infinite in size because your system can only process numbers up to a certain size. BASIC graphics uses data of type SHORT; therefore, the largest absolute value of x or y that can be input is approximately 3.4×10^{38}. That is, you can't plot any coordinates greater than this value.

Think of the display as a window to this plane. You can look at the whole coordinate system or a very small part of it at any time through this window. When you turn on the computer, the lower left-hand corner of the display is (0,0), and the upper right-hand corner is (133,100). The viewable upper right hand corner for an Enhanced Graphics Display is (133,90). The upper edge of the FRAME will not be visible. Refer to appendix B.

Enter and run this program:

```
10 GINIT
20 GRAPHICS ON
30 FRAME
40 WAIT 5
50 END
```

A frame outlined in white will appear on the display. This frame surrounds the entire plotting surface, and is smaller than the CRT screen. Any coordinates to which you can actually plot are within this frame. You can execute graphics output statements that are beyond the edge of the display, but no primitives are output. Placing a frame around the usable plotting area can help when composing a picture.

Finding the Current Position. Besides knowing the plotting area that you have to work with, you need to know where the current position is. This is the point on the display relative to which subsequent graphics are positioned. To find it, use the WHERE command. Enter and execute the following program:

```
10 WHERE X,Y
20 PRINT "X =";X,"Y =";Y
30 END
```

X returns the x coordinate of the current location and Y returns the y coordinate. Right now, X and Y should equal 0 because whenever GINIT is executed, the current position is (0,0).

Changing the Current Position. The next step is to place the current position where you want to start drawing. To do this, use the MOVE statement. Execute the command:

```
MOVE 50,50
```

Although nothing looks different on the display, you moved the current position to the point (50,50). Use WHERE to see that the current position has changed.

You can use any expression in the range of SHORT values in MOVE. (In fact, almost all the graphics statements can work with expressions in the range of SHORT values.) After a GINIT, the resolution of the CRT is such that coordinates have two significant decimal places. Other plotters have different resolutions; you have to experiment to determine these.

You may want to displace the current position by a specific amount. In this case, instead of working out the coordinates of the new position, you can specify an incremental movement. You do this with the IMOVE (Incremental MOVE) command. Execute:

```
IMOVE 10,5
```

You've moved the current position by 10 units along the X axis and 5 units along the Y axis to the point (60,55). Again, WHERE can confirm this.

Digitizing the Current Position. Often you can see where you would like to move the current position, but you can't tell what the exact coordinates of the point are. To determine the coordinates of a location on the display, use digitizing.

Here is how to do this.

- Put a crosshair on the display.
- Move the crosshair to the point on the display that you want to be the current position by using the arrow keys of the keyboard.
- Tell the system to return the coordinates of the crosshair's position.

You can then use these coordinates to move the current position to that point. Execute the commands:

```
TRACK CRT IS ON
DIGITIZE X,Y
```

The left and bottom edges of the display have a bright white line. The TRACK...IS ON command sets a full-screen crosshair at the point (0,0). TRACK CRT IS ON tells the system that you want to "track" or mimic the keyboard arrow keys on the internal CRT with a crosshair.

Use the arrow keys on the keyboard or the optional mouse to move the crosshair to the point you wish to digitize. Digitize tells the system that you want to digitize or record the arrow key's position. At this point, you can still move the crosshair around if you wish. DIGITIZE doesn't store the coordinates until the next time (Enter) is pressed.

When you have the crosshair positioned at the desired point, press (Enter) again to get the system to actually digitize the point and place the coordinate values in the variables X and Y. X has the x coordinate and Y has the y coordinate. When the system is waiting for you to press (Enter) to digitize a point, the run light looks like an asterisk.

You can move to the point you've just found by executing the command:

```
MOVE X,Y
```

This form of digitizing only works for the internal CRT and keyboard.

The crosshair does not disappear after you're done digitizing. You can still move it around and digitize another point or move the crosshair out of the way.

If your display is generated by a program, you can turn the crosshair off with:

```
TRACK CRT IS OFF
```

then execute:

```
GCLEAR
```

and then regenerate the display by running the program again without the TRACK CRT IS ON statement.

Creating Graphics

Drawing Lines

At this point, you should be able to move the current position to any point. This section explains how to draw lines on the CRT.

Execute these commands:

```
GINIT

MOVE 50,50

DRAW 60,60
```

You have a line on the display from the point (50,50) to the point (60,60). You can draw any line by first moving the current position to the starting point and then drawing to the end point. In addition, the current position is updated to the point (60,60). Use WHERE to see this.

If you want to draw a line of a certain length but you don't want to figure out the coordinates of the end point, use the IDRAW (Incremental DRAW) command. Execute:

```
IDRAW 10,10
```

This command draws a line to a point 10 units along the X axis and 10 units along the Y axis from the current position. It also updates the current position by the specified increments.

See if you can recreate the following picture using 21 statements. You may use any of the statements presented up to this point.
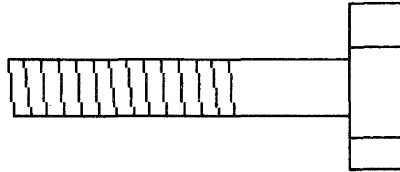
Load and run the file "BOLT" from the Manual Examples disc. You can list the file on your CRT or printer to compare your version.

As mentioned before, you can plot to points beyond the edge of the display, but they do not appear. If part of the line is within the display area, that part is output. As an example, load and execute the file "BIGLINES" from the Manual Examples disc. You should see the figure below.
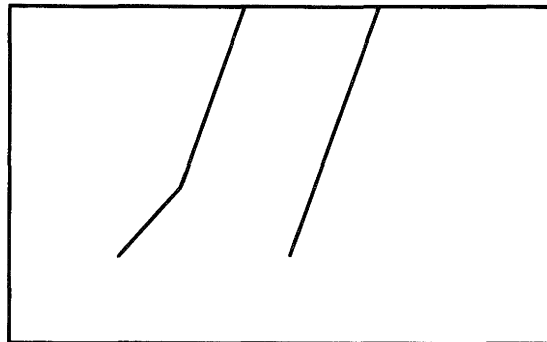
All the graphics output is handled in a similar manner. Those points within the display area are output. Those points outside the display area are not shown, but the current position is updated.

DRAW, IDRAW, IMOVE and MOVE have an additional effect beyond moving the current position and drawing a line. They also determine whether the physical pen is up or down (that is, touching the plotting surface). For a CRT, when the pen is "down," a dot appears. When the pen is "up," no dot is created.

One way to control whether the pen is up or down on a plotter is to use IMOVE AND IDRAW.

```
IMOVE 0,0
```

lifts the pen, but does not change the current position.

```
IDRAW 0,0
```

lowers the pen, but does not change the current position.

Another way to raise the pen is by executing the PENUP statement.

```
PENUP
```

The WHERE statement has an additional parameter with which you can determine the status of the pen. It is a string variable whose contents signals whether the pen is up or down and whether it is within the display area. The string also returns other indormation that is explained later. For example, execute:

```
10 WHERE X,Y,STATUS$
20 PRINT "X =";X,"Y =";Y,"STATUS =";STATUS$
30 END
```

STATUS$ is a string which looks like:

```
1,2
```

The first digit describes the pen's vertical position (0 = up, 1 = down). The second digit describes the pen's horizontal position within the display area (0 = outside of the display area, 1 or 2 = inside the display area).

When you use a paper plotter make sure that the pen is lifted if it is going to rest in one spot for very long; otherwise, an ink blot occurs.

Scaling

Some graphics may not show much information. There may not be enough variation in the data as presented. For example, load and run the file "SCALE" from the Examples disc.

Probably the first reaction you had when looking at the plot was "That doesn't show me anything...". That's true; it doesn't show much information. There are two reasons for this. The first is that there is not enough variation in the curve; it's too straight to show anything. The second is that it is not centered.

Both of these problems can be remedied by scaling. In this context, scaling could be defined as "defining the values the computer considers to be at the edges of the plotting surface." By definition, the left edge is the smaller X, the right edge is the larger X, the bottom is the smaller Y, and the top is the larger Y. Thus, any point you plot that falls into these ranges will be visible.

There are two statements available to define your own values for the edges of the plotting surface . The first one we'll deal with is SHOW, which forces X and Y units to be equal. Since the X and Y units are identical, the SHOW statement centers the specified area in the plotting area. This is called isotropic scaling, and it is often desirable. For example, when drawing a map, you will probably want one mile in the east-west direction to be the same size as a mile in the north-south direction. Here is an example of SHOW:

```
SHOW 0,100,16,18
```

This causes the plotting area to be defined such that there is a rectangle in that plotting area whose minimum X is 0, maximum X is 100, minimum Y is 16, and maximum Y is 18, *using isotropic units.* As mentioned above, isotropic means that one unit in the X direction is equal to one unit in the Y direction. Hence, if the plotting area were square, the above SHOW statement would define the minimum X to be 0, maximum X to be 100, minimum Y to be -33 (not 16) and maximum Y to be 67 (not 18). The reason for this is that allowing whatever extra room it needs to insure that that rectangle is completely contained in the plotting area. There will be extra room in either the X or Y direction, but not both.

Since you (the user) were defining unit sizes with the SHOW statement, you were working with User-Defined Units (UDUs). Both the SHOW statement and the WINDOW statement (covered next) specify user-defined units. Load and run the file "SCALE2".

As you can see, the SHOW statement takes care of centering the curve on the screen, but since the range of X values is so much larger than the range of Y values (0 to 100 versus 16 to 18), it still does not give us enough resolution to see what the data is doing. Isotropic scaling is desirable in many cases. In many other cases, however, it is not. If this example shows the graph of the voltage from a sensor versus time, it makes no sense to force seconds to be just as "long" as volts. Since the data types are not equal, it would be better to use unequal, or anisotropic, scaling. You can do this with the other scaling statement: WINDOW. This will not force X units to be equal to Y units. Now load and run the file "SCALE3".

This plot looks much better than the last one; you can easily see variations in the data. To test how the Y axis range 15–19 affects relative variations in the data, list the program in file "SCALE3" and change line 30 to WINDOW 0,100,30,50 and change line 50 to PLOT X,RND + 40. Run the program again and note that the line is less ragged.

There is still one problem, though. You can see *relative* variations in the data, but what are the units being used? That is, is the height of the curve signifying differences of microvolts, millivolts, megavolts, dozens of volts, or what? And you probably wouldn't want the text (explaining units, etc.) to be written in the same area that the curve is in, as it could obstruct part of the curve. Therefore, you need to be able to specify a subset of the screen for plotting the curve, and put explanatory notes outside this area. The next section tells you how to do this.

Defining a Viewport

A viewport is a subset of the plotting area. This is called the soft clip area, and it is delimited by the soft clip limits. Clip, because any line segments which attempt to go outside these limits are cut off at the edge of the subarea. Soft, because you can override these limits by turning off the clipping with the CLIP OFF statement (more about this later). There are hard clip limits also, and these are defined to be the absolute limits of the plotting area. Under no circumstances can a line be drawn outside of these limits. There is no way to override the hard clip limits, as you could with soft clip limits.

GDUs and UDUs. There are two types of units used to define viewport limits. These are UDUs (User-Defined Units) and GDUs (Graphics Display Units). In order for viewports to be predictable, they must always be specified in the same units. Since UDUs are subject to change, you should use GDUs when specifying the limits of a VIEWPORT statement. GDUs are fixed for the CRT, so a viewport is associated with the screen, rather than the graphical model used in your program.

Unless you specify otherwise, the screen (but *not* necessarily an external plotter) is considered to have the following expanse: in the X direction, 0 through 133.444816054; in the Y direction: 0 through 100. These are GDUs. The lower left of the plotting area is *always* 0,0. The length of a GDU is defined as "One percent of the shorter edge of the plotting area."

Since the height of the screen is shorter than the width of the screen, the shorter edge is in the Y direction, therefore, Ymax in GDUs is 100. If the screen had been higher than it is wide, Xmax in GDUs would have been 100. That was the easy part. Once you've decided which edge is shorter, and thus defined the units along that edge, you need to find out how many GDUs in extent the *longer* sides are. For now, just observe that the GDU limits are 0 to 133.444816054 in X, and 0 to 100 in Y.

Note   👉   If you are using an Enhanced Graphics Display, only 0 to 89.74 GDUs are viewable in Y.

Specifying the Viewport. The VIEWPORT statement defines the extent of the soft clip limits in GDUs. It specifies a subarea of the plotting surface which acts just like the entire plotting surface, except that you can draw outside the subarea if you turn off clipping. Load and list file "SCALE4" from the Manual Examples disc. The VIEWPORT statement in this program specifies that the lower left-hand corner of the soft clip area is at 10,15 and the upper right-hand corner is at 120,90. This is the area which the WINDOW statement affects. Also note line 40; the FRAME statement. This draws a box around the current soft clip limits. It is used in this example so you can see the area specified by the VIEWPORT statement. Now run the program to see the result.

Labelling a Plot. With the inclusion of the VIEWPORT statement, you have enough room to include labels on the plot. Typically, in a plot like this, there is a title for the whole plot centered at the top, a Y-axis title on the left edge, and an X-axis title at the bottom.

You can use the LABEL statement to write text onto the graphics screen. You can position the label by using a MOVE or PLOT statement to get to the point at which you want the label to be placed. It is the lower left corner of the label which ends up at the point to which you moved. In other words, you move to the position on the screen at which you want the lower left corner of the text to be placed.

Load and run the file "LABELS" from the Manual Examples disc. Notice that the Y-axis label on the left edge of the screen is created by writing one letter at a time. You only need to move to the position of the first character in that label because each label statement automatically terminates with a carriage return/linefeed. This causes the pen to go one line down, ready for the next line of text.

Now you know what you are measuring—voltage vs. time—but you still do not know the units being used. You need an X-axis and a Y-axis labelled with numbers in appropriate places. You can use the AXES statement to accomplish this.

Axes and Tick Marks. You can use the AXES statement to draw X and Y axes and short lines, perpendicular to the axes, to indicate the spacing of units. These short lines are called tick marks. The axes may intersect at any point you desire. The tick marks may be any distance apart, and you can select the "major tick count" for each axis. The major tick count is the total number of tick marks drawn for every large one. This makes it convenient to count by fives or tens or whatever you chose the major tick count to be. And finally, you can specify how long you want the major tick marks to be. This is measured in GDUs. Enter the following program:

```
10  GINIT
20  FRAME
30  AXES 5,10,50,50,3,3,3
40  WAIT 5
60  END
```

When you run this program, you should see the figure below:
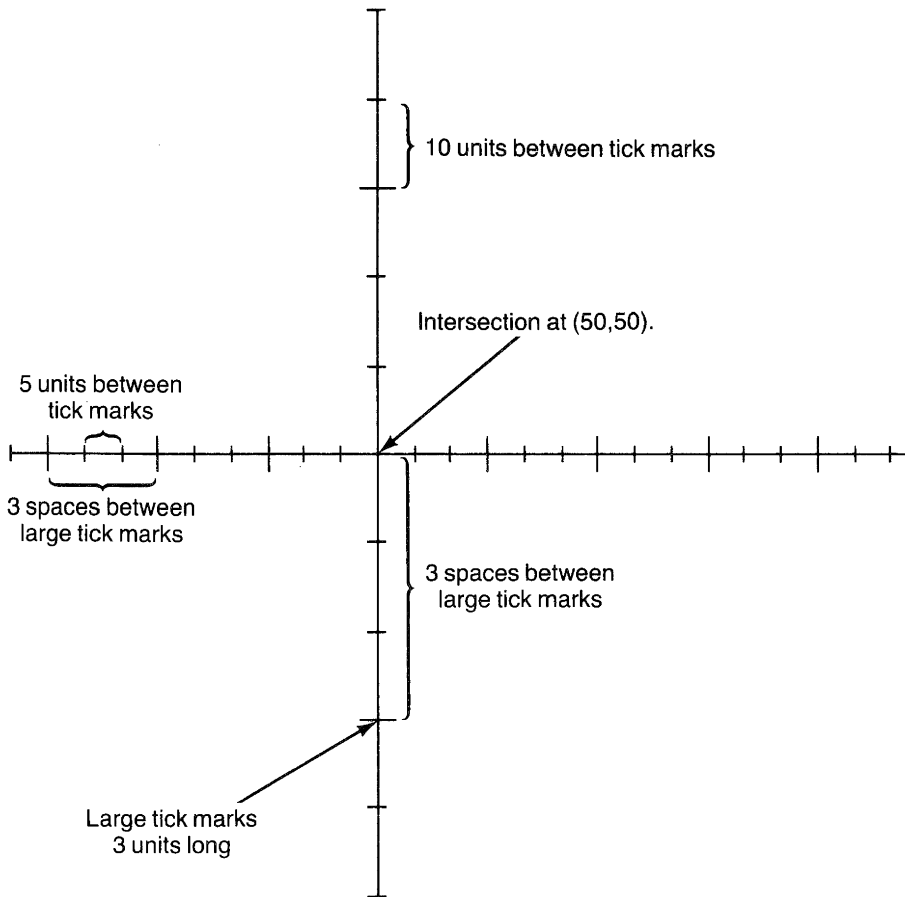
In the axes statement, the first parameter specifies the distance between tick marks along the horizontal (x) axis, and the second parameter specifies the distance along the vertical (y) axis. The third and fourth parameters specify the intersection point of the axes. The fifth and sixth parameters specify the number of spaces between the large tick marks, and the last parameter specifies the size of the large tick marks. The small tick marks are drawn half the size of the large tick marks.

Now load the file "LABELS" again, add the AXES statement shown below to this program, and run it to see the difference.

```
145   AXES 1,.1,0,15,5,5,3
```

Grids. You can also create a full grid pattern. Enter and run the following program:

```
10   GINIT
20   FRAME
30   GRID 5,10,50,50,3,3,3
40   WAIT 5
50   END
```

When you run this program you should see the following:
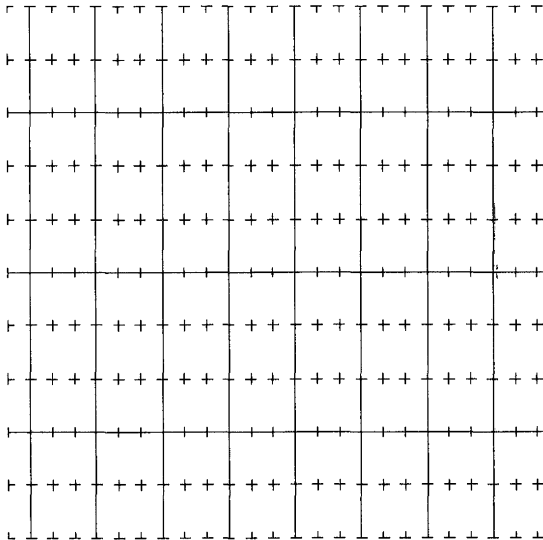
Some of the parameters have slightly different meanings in a GRID statement than in an AXES statement. The first two still represent the distance between tick marks in the horizontal and vertical directions respectively. The next two parameters specify the intersection point of two lines in the grid. The fifth and sixth parameters still specify the number of spaces between large tick marks for each axis. The last parameter still specifies the length of the tick marks.

Like frames, axes and grids are always parallel to the edges of the display. Axes and grids are affected by the line type and pen type.

Load and run the file "AXES" from the Manual Examples disc to see various kinds of axes and grids.

You can stop this program at any time by pressing STOP.

Other Ways to Draw or Move

There are three other ways to draw or move. The first is using the PLOT statement. With this statement, you specify the point to which the pen moves and also whether the pen is up or down before or after it is repositioned. Here is a PLOT statement.

```
PLOT 10,40,2
```

The first two items in the statement are the x and y coordinates, and the third item is an optional pen control value.

The following table shows how the pen control value affects the output.

Pen Control Value	Meaning
Negative and even (−2, −4, −6, ...)	Raise the pen before repositioning it.
Non-negative and even (0, 2, ...)	Raise the pen after repositioning it.
Negative and odd (−1, −3, −5, ...)	Lower the pen before repositioning it.
Positive and odd (1, 3, 5, ...)	Lower the pen after repositioning it.

If no pen control is specified, 1 is assumed.

IPLOT (Incremental PLOT) is the second way to move or draw. It is similar to PLOT. The difference is that with IPLOT each repositioning is incremental like IMOVE OR IDRAW. The pen control values cause actions similar to those in PLOT.

RPLOT (Relative PLOT) is the third way to move or draw. It is similar to PLOT in that the RPLOT parameters are displacements from an origin, but they are displacements from a local origin. A local origin is a temporary origin for all consecutive RPLOT statements. Each coordinate given in an RPLOT statement is measured from the local origin. The local origin is defined as the current position when the first RPLOT is executed. When you stop executing consecutive RPLOTs, that is, when a graphics output statement other than an RPLOT statement is executed, the local origin ceases to exist.

The next program is an example of how RPLOT works. Load the file "STARS" and list the program. Step through it slowly and determine what happens with each statement before executing it.

Notice how the local origin is set by a MOVE and a series of IMOVEs. The pattern is actually drawn by repeating the subroutine, Rplot. RPLOT is particularly useful when drawing the same series of lines in different spots on the CRT.

PLOT, IPLOT, and RPLOT are useful for controlling the pen with a formula or variable. For instance, you might want to create a variable Pen_status. If Pen_status equals −2, no line is drawn, but the current position is updated.

Erasing Lines

Now that you can draw lines, you're probably wondering how to erase them. GCLEAR clears the entire graphics area of the CRT screen. To eliminate one specific line or portion of a line, use the PEN statement. The PEN statement has the form:

```
PEN Pen_number
```

where Pen_number is a numeric expression which specifies the pen to use.

The PEN statement gives a choice of "pen" with which to plot. The default pen type is 1. This is the pen type that creates the white line you have seen so far.

A pen selection of −1 sets the pen type to erase white. Output statements are then executed with that color. If the primitive crosses over a point on the display which is white, that point becomes black.

Note that any output statement using this pen value not only erases lines created by DRAW or IDRAW, but also erases part of a frame or any other graphics output created with PEN 1. Obviously, a pen type of −1 is only usable with a plotter that can erase part of its display, such as a CRT. Plotters that ouput on paper ignore a PEN −1 statement.

Line Attributes

What is it about lines that distinguishes them from other lines? The color of a line can set it apart from other lines; so can the pattern used to draw it (for example, dashed or dotted). These distinguishing traits are known as attributes of the line. Your system provides a number of attributes for graphics primitives.

Pen Types

The PEN statement presented in the previous section does not directly create graphics output but does affect the appearance of graphics output. For this reason, it is an "attribute" statement.

On a monchrome CRT, a pen type > 0 makes the pen color "white". A pen type < 0 makes the pen color "erase white". A pen type of 0, in effect, disables the pen; no lines or erasures occur when a DRAW or other output statement is executed. However, it does not disable updating of the current position; that is, if you execute a DRAW, no line is drawn, but the current position is now at the coordinates specified in the DRAW.

PEN can also specify other colors if you have a color CRT or other pen stalls on a paper plotter. The "Color" section provides more information about these pen types.

Load and run the file "PENDEMO" from the Manual Examples disc.
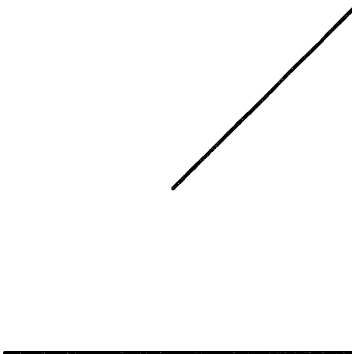
Line Types

To distinguish between lines, use different line types such as dashes or dots. The LINE TYPE statement gives a number of choices. The pattern of a line is considered an attribute of that line. Load and run the file "LINETYPES" from the Manual Examples disc.
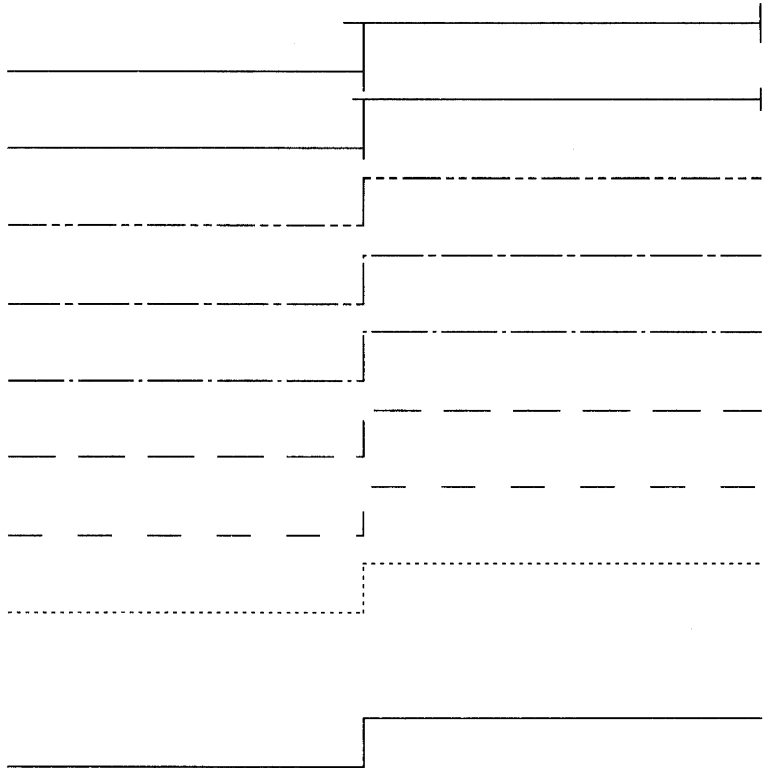
This shows the available line types. It also shows how the lines look when drawn straight or around corners.

Creating Simple Shapes

BASIC Graphics has specific statements to create many kinds of regular polygons quickly and easily.

Rectangles. The simplest polygon is probably the rectangle created used the RECTANGLE statement. Enter and execute the following program

```
10   GINIT
20   MOVE 20,20
30   RECTANGLE 10,30
40   WAIT 5
50   END
```

The first parameter is the width of the rectangle and the second parameter is the height of the rectangle. In this case, you've drawn a rectangle 10 units wide and 30 units high.

RECTANGLE is another statement which has, in effect, a local origin. The rectangle is drawn with the current position (local origin) in the lower-left corner. In the example above, the local origin is at 20,20. The sides of the box are parallel to the sides of the CRT. This is only the default form. You can use the PDIR (Plot DIRection) statement to rotate the rectangle to any angle about the Z axis. Its form is:

```
PDIR Angle
```

where Angle is the amount the polygon is rotated in degrees, radians or grads.

As an example, load and run the file "PDIRDEMO" from the Manual Examples disc. You should see the following:
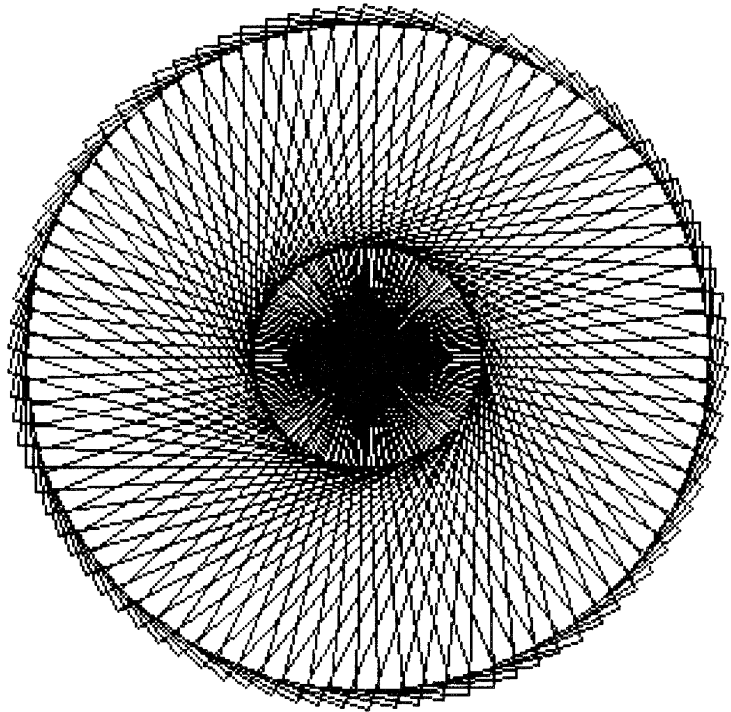
Notice that the rotation is about the local origin (lower-left corner) of the ractangle. This means that when placing a rectangle on the display, you should note where the lower-left corner should go, not where the center of the rectangle should go.

The RECTANGLE statement has two other options. They are FILL and EDGE. If you specify FILL, the rectangle is filled to create a solid block. If you specify EDGE, the rectangle edge is drawn with the current pen and line type. Load and run the file "FILLEDGE" from the Manual Examples disc. You should see the following:
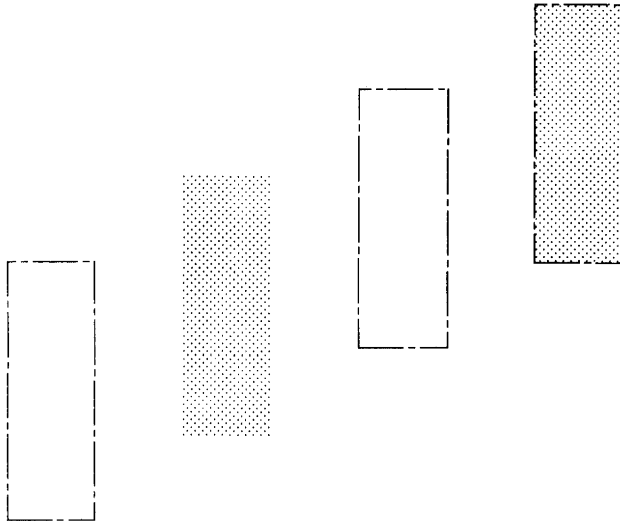
FILL Attributes. shade of the fill color to various degrees of grey using the AREA COLOR or AREA INTENSITY statements. These statements are fully explained in the "Color" section, but the following paragraphs explain how to use them with a monochrome CRT.

The AREA COLOR statement defines the fill color based on a hue, a saturation and luminosity. Only the third parameter, the luminosity parameter, has any effect on a monochrome CRT. For example:

```
AREA COLOR .1,.1,.1

AREA COLOR .5,.6,.1

AREA COLOR .7,.2,.1
```

All specify the same shade of grey. The luminosity parameter specifies the approximate percentage of pixels to be "on" in the filled area. The previous three statements all turn on 10% of the pixels in the fill area. The range of the luminosity is from 0 thru 1.

The AREA INTENSITY statement defines the fill color based on the largest of the three paramenters in the statement. For example,

```
AREA INTENSITY .1,.3,.4

AREA INTENSITY .4,.2,.01

AREA INTENSITY .2,.4,.03
```

All specify the same shade of grey. Again, the largest parameter specifies the approximate percentage of pixels to be "on" in the filled area. The previous three statements all turn on 40% of the pixels in the fill area. Again, the range of the parameters in the statement is 0 thru 1.

Polygons. To create more general polygons use the POLY-
GON statement. Enter and execute the following:

```
10   GINIT
20   GRAPHICS ON
30   MOVE 50,50
40   POLYGON 10,6
50   WAIT 5
60   END
```

The first parameter of the POLYGON statement is the radius
of the polygon and the second is the number of sides. Thus,
POLYGON 10,6 creates a six-sided polygon with a radius of
10 centered about the point (50,50).

The polygon is output with the first vertex at an angle of 0
degrees from the X axis.

The POLYGON statement forces the shape to be a closed
polygon as opposed to an open polygon.

You may specify the number of edges to draw and the POLY-GON statement then closes the shape. For example, load and run the file "POLYGON6" from the Manual Examples disc. You should see the following:
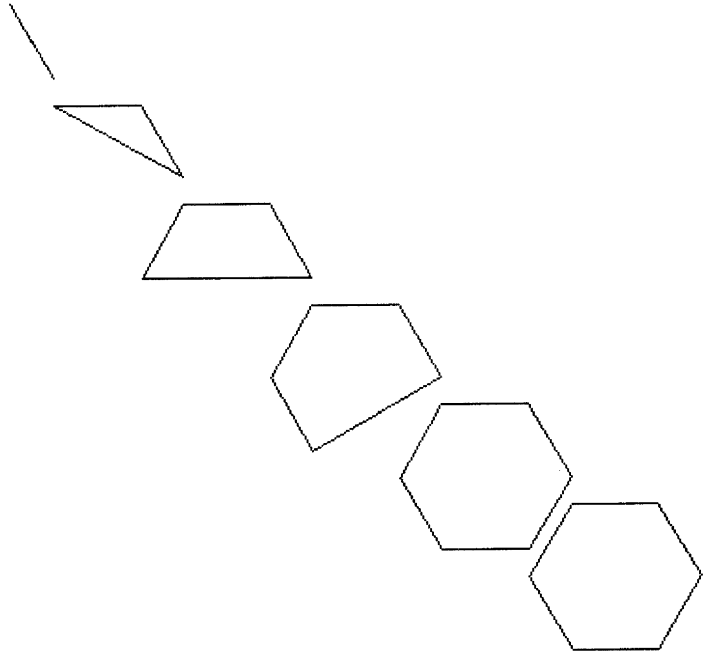
Whether the pen is up or down before the polygon is drawn
makes a difference. Load and run the file "POLYGON4 from
the Manual Examples disc. You should see the following:
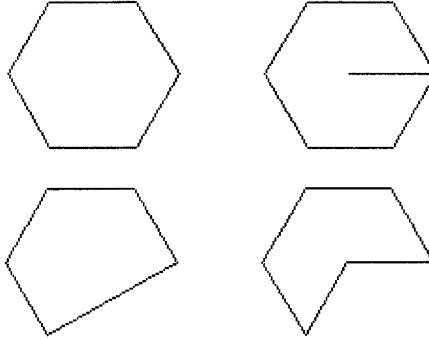
If the pen is down when this statment is executed, the first
line is from the pen's starting position out to the first vertex
and the last line is from the last vertex back to the pen's origi-
nal position. If the pen is up when the statement is executed,
the last line is from the last vertex back to the first. In either
case, the current position is unchanged.

POLYGON also has the FILL and EDGE options like
RECTANGLE.

Sometimes you don't want a closed polygon. In these cases, use the POLYLINE statement. Load and run the file "POLYLINE"
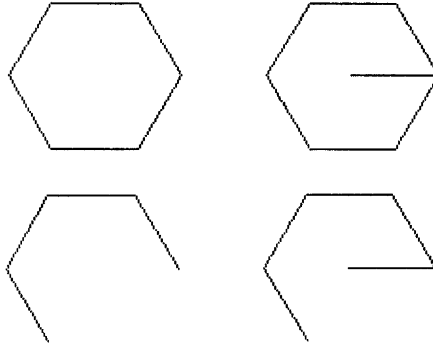
You can see that polylines are similar to polygons, but they don't have to be closed.

The general rule is: if the pen is down when this statement is executed, the first line is from the pen's starting position out to the first vertex. If the pen is up when this statement is executed, the figure starts at the first vertex. The figure always stops at the last vertex. The current position remains at its original location.

PDIR also affects polygons and polylines. The local origin is at the center of the polygon/polyline so the rotation is about the center of the object. Load and run the file "CIRCLES2" from the Manual Examples disc to see PDIR's effect.
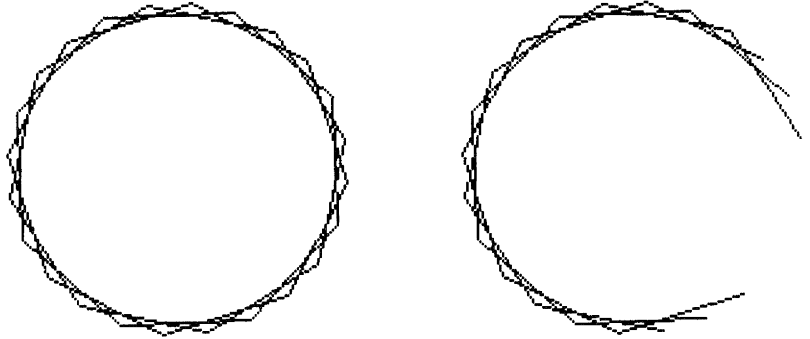
Additional Pen Control

There are additional pen control values available when using PLOT with an array that can't be used with PLOT x,y,z. These help you to make more complex drawings than are possible with PLOT x,y,z. For instance with PLOT x,y,z, a pen control value which is a positive odd integer lowers the pen after repositioning it. However, when using PLOT with an array, only a pen control value of 1 causes this. A pen control value of 3 tells the system that the x coordinate value is to be interpreted as a pen number. Thus, you can switch pens while plotting from an array. Here is a complete table of all the pen control elements and how they are interpreted.

X Coord. Element	Y Coord. Element	Z Coord. Element	Pen Control Element	
This element is interpreted as:	This element is interpreted as:	This element is interpreted as:	If this element is:	Action:
x coord.	y coord.	z coord.	negative even number (-2, -4, -6, ...)	Raises the pen and then repositions it.
x coord.	y coord.	z coord.	negative odd number (-1, -3, -5, ...)	Lowers the pen and then repositions it.
x coord.	y coord.	z coord.	0 or 2	Repositions the pen and then raises it.
x coord.	y coord.	z coord.	1	Repositions the pen and then raises it.
pen number	n/a	n/a	3	Selects this pen.
line type	repeat length	n/a	4	Selects this line type and repeat length.
fill color (see the following text)	n/a	n/a	5	Selects this fill color.
n/a	n/a	n/a	6	Starts a polygon with FILL.
n/a	n/a	n/a	7	End of a polygon.
row number	n/a	n/a	8	End of plotting data.
n/a	n/a	n/a	9	Index of End-of-plotting-data row
n/a	n/a	n/a	10	Starts a polygon with EDGE.
n/a	n/a	n/a	11	Starts a polygon with FILL and EDGE.
n/a	n/a	n/a	12	Frames the current display area.
n/a	n/a	n/a	>12	Row Ignored.

The fill color gives you the ability to change the color of a filled polygon. The exact explanation of the process is covered in the "Color" section of this chapter.

Now load and list the "SHIP" from the Manual Examples disc. The DATA lines in this program are treated as an example array. A step-by-step explanation of the data elements as they would be interpreted in a two-dimensional PLOT is shown in the matrix that follows.

0	0	12	Frames the display.
40	10	−2	Raises the pen and moves it to (40,10).
11	0	5	Selects the fill color defined by the value 11. The 0 is ignored.
12	34	11	Signals the start of a polygon that is filled and has an edge. The values 12 and 34 are ignored. Every consecutive line from this point on is considered part of the polygon until the figure is closed. The current position is still (40,10).
100	10	−1	Lowers the pen and draws a line from (40,10) to (100,10).
110	20	−1	Keeps the pen down and draws a line from (100,10) to (110,20).
15	20	−1	Keeps the pen down and draws a line from (110,20) to (15,20).
40	10	−1	Keeps the pen down and draws a line from (15,20) back to (40,10). After this line is completed there is a closed figure so it is filled and the edge is kept.
0	0	7	Signals the end of the polygon. If this line is not included and any DRAW actions are included before a MOVE, those DRAWs are considered part of the polygon. Thus, when a MOVE is executed the DRAWs are closed off to create a polygon.
65	20	−2	Raises the pen and moves it to (65,20).

0	0	6	Signals the start of another polygon with the FILL attribute. The zeroes in the first two columns are more appropriate than the values left in the first two elements of row 4.
64	80	−1	Draws a line from (65,20) to (64,80).
63	20	−1	Draws a line from (64,80) to (63,20).
0	0	7	Signals the end of a polygon so the two lines are closed off to form a triangle and then the polygon is filled.
64	80	0	Draws a line from (63,20) to (64,80) and then lifts the pen
1	0	5	Selects the fill color defined by the value 1.
0	0	6	Signals the start of a polygon with FILL.
50	25	−1	Lowers the pen and draws a line to (15,20).
15	20	−1	Keeps the pen down and draws a line to (15,20).
64	80	−1	Keeps the pen down and draws a line to (64,80).
0	0	7	End of the polygon.
64	75	−2	Moves the pen to (64,75).
0	0	6	Start of another polygon with FILL.
70	25	−1	Lowers the pen and draws a line from (64,75) to (70,25).
110	20	−1	Keeps the pen down and draws a line to (110,20).
64	75	−1	Keeps the pen down and draws a line to (64,75).
0	0	7	End of the polygon.

Now run the program to see the results:
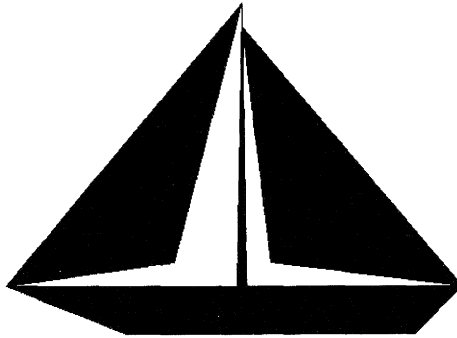
Using Graphics Effectively

The last section discussed the more elementary graphics operations. This section will present more detailed information on those statements and introduce several other graphics operations.

Most of the demonstration programs in this section are stored on the Manual Examples disc which was shipped with this manual. You are encouraged to load and run these programs as you are reading the manual, as this will make understanding the concepts much easier.

More on Labelling a Plot

There are three statements that complement the LABEL statement; they expand its capabilities greatly.

The first is CSIZE, which means *character size*. CSIZE has two parameters: character cell height (in GDUs) and aspect ratio. The height measures the character *cell* size. A character cell contains a character and some blank space above, below, left of, and right of the character. This blank space allows packing character cells together without making the characters illegible. The amount of blank space depends, of course, on which character is contained in the cell.

The first of these small programs shows how the CSIZE statement changes the size of characters. You may load this program from file "CSIZE" on the Manual Examples disc.

When you run the program you should see something like this:
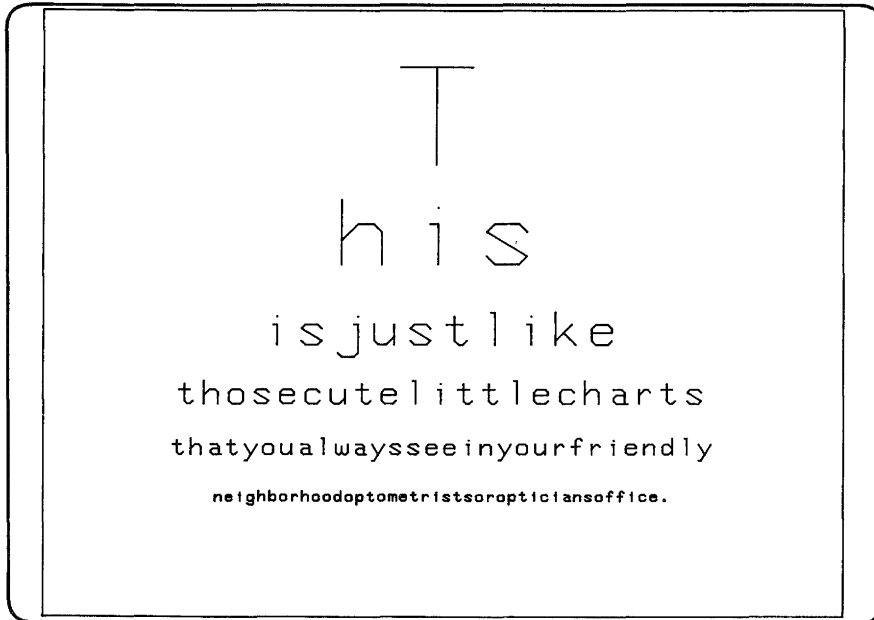
The FOR-NEXT loop writes lines of text on the screen with different character sizes. The DATA statements contain both pieces of information. Incidentally, notice also the WINDOW statement. It specifies a Ymin *larger* than the Ymax. This causes the top of the screen to have a lesser Y-value than the bottom. This is perfectly legal.

The next program deals with the relationship between the size of the character, per se, and the size of the character cell—that rectangle in which the character is placed. This program is on file "CHARCELL" on the Manual Examples disc.

Size of Character in Character Cell
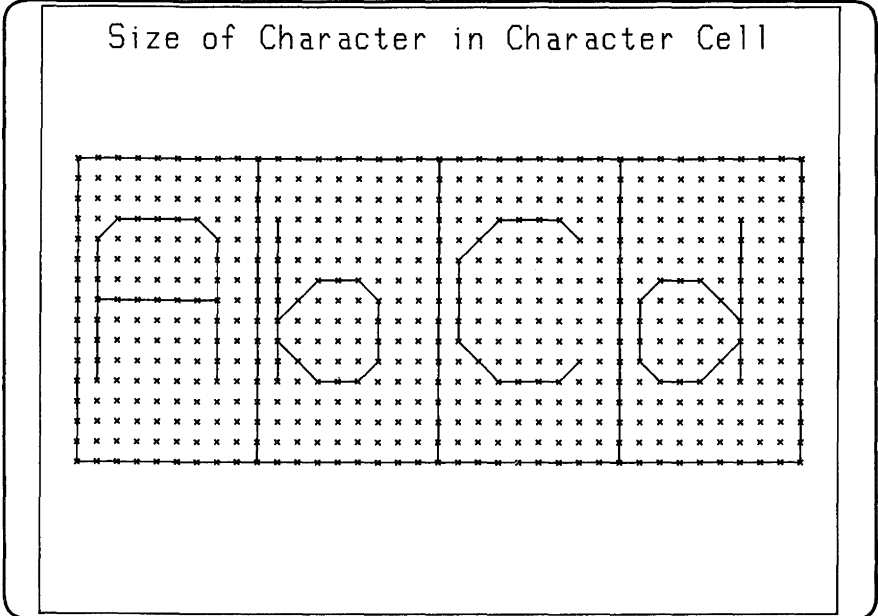
As the diagram shows, a character is drawn inside a rectangle, with some space on all four sides. The rectangle's height is specified by the CSIZE statement, and is measured in GDUs. The rectangle's width (also measured in GDUs) is the height multiplied by the aspect ratio. This rectangle is subdivided into a grid of 9 wide by 15 high. Characters are drawn in this framework, called the symbol coordinate system. Of course, the little Xs in the plot above are not drawn when you label a string of text; they are there solely to show the position of the characters within the character cell.

Again, character cell height is measured in GDUs, and the definition of aspect ratio for a character is identical to the definition of aspect ratio for the hard clip limits mentioned earlier: the width divided by the height. Thus, if you want short, fat letters, use an aspect ratio of 1.5 or larger. If you want tall, skinny letters, use an aspect ratio less than about 0.5.

CSIZE 3   Cell 3 GDUs high, aspect ratio .6 (default).

CSIZE 6,.3   Cell 6 GDUs high, aspect ratio .3 (tall and skinny).

CSIZE 1,2   Cell 1 GDU high, aspect ratio 2 (short and fat).

Note that you do not have to specify a second parameter (the aspect ratio) in the CSIZE statement. This defaults to 0.6.

The second statement you need is LORG, which means label origin. This lets you specify which point on the label ends up at the point moved to before writing the label. You may load the following program from file "LORG" on the Manual Examples disc.

```
LORG 1 =                          ₓTEST
                                  ×

LORG 2 =                          ×TEST

LORG 3 =                          ×
                                   TEST
LORG 4 =                       TEST
                                  ×

LORG 5 =                       TEST

LORG 6 =                          ×
                               TEST
LORG 7 =                    TEST
                                ×

LORG 8 =                    TESTₓ×

LORG 9 =                              ×
                            TEST
```
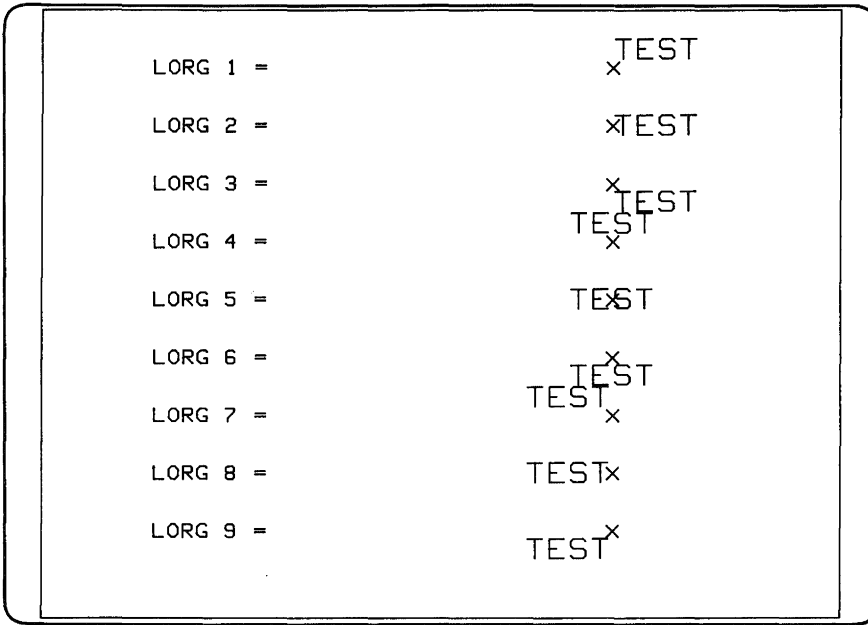
The x's indicate where the pen was moved to before labelling the word "TEST". This diagram shows that, for example, if LORG 1 is in effect and you move to 4,5 to write a label, the lower left of that label would be at 4,5. This automatically compensates for the character size, aspect ratio, and label length. It makes no difference whether there is an odd or even number of characters in the label. If LORG 6 had been in effect, and you had moved to 4,5, the center of the top edge of the label would be at 4,5. You can readily see how useful this statement is in centering labels, both horizontally and vertically.

The third statement you need to know is LDIR, meaning label direction. This specifies the angle at which the subsequent labels will be drawn. The angle is specified in the current angular units, and is either DEG (degrees) or RAD (radians). For example, assuming degrees is the current angular mode:

LDIR 0   Writes label horizontally to the right.

LDIR 90   Writes label vertically, ascending.

LDIR 14   Writes label ascending a gentle slope, up and right.

LDIR 180   Writes label upside down.

LDIR 270   Writes label vertically, descending.

Load and list the file "LDIR" on the Manual Examples disc. You'll note that LORG 2 was specified (line 70), and this remained in effect for many LDIRs. Each label is centered on the left edge (relative to the *label*, remember). Now run the program and you should see the following:
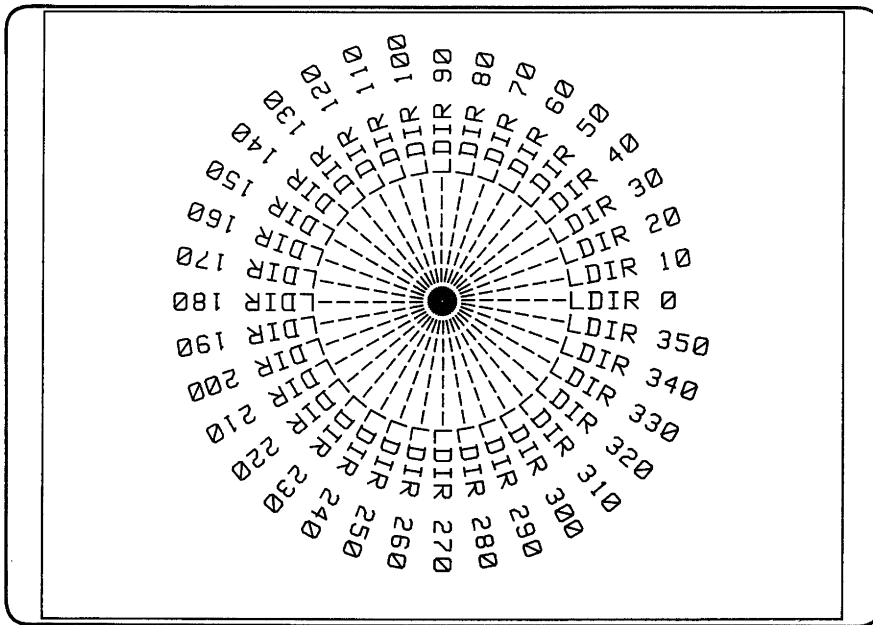
The label origin specified by LORG is relative to the *label*, not the plotting surface, and it is independent of the current label direction. For example, if you have specified

```
LORG 3
DEG
LDIR 90
MOVE 6,8
```

and then write the label, it is written going straight up, not horizontally. Therefore, it is the upper left corner of the label which is at point 6,8 *relative to the rotated label*. Relative to the plotting device, however, it is the lower left corner of the label which is at 6,8 (in this example) because the label has been rotated.

Now that the prerequisites have been taken care of, we can discuss the statement which actually causes labels to be written: LABEL. LABEL takes into account the most recently-specified CSIZE, LDIR and LORG when it writes a label. You must position the label to get to the point at which you want the label to be placed. You can use MOVE to accomplish this.

All four statements have been utilized in the following update to our progressive plotting program. You may load this program from file "SINLABEL" on the Manual Examples disc.

VOLTAGE VARIANCE

Voltage

Time (seconds)
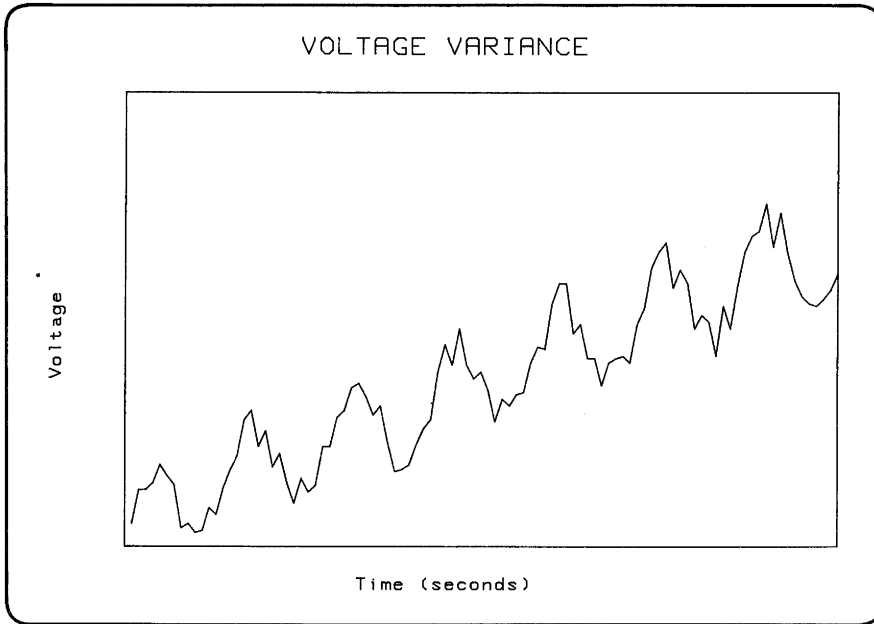
Notice that the title of the graph, "VOLTAGE VARIANCE", is now displayed in bold face. This effect was achieved by plotting the label several times, moving the label origin just slightly each time. Notice lines 60 through 90. The loop variable, I, goes from −.3 to .3 by tenths. This is the offset in the X direction (in GDUs*) of the label origin.

Since this is being labelled with LORG 6 in effect, the label origin (the point moved to immediately prior to labelling) represents the center of the top edge of the label. This method can also be used for offsetting in the Y direction. Or, offset both X and Y. This will give you characters which are thick in a diagonal direction, which makes them look like they are coming out of the page at you. However, a more typical bold-face is produced by offsetting only in the X direction.

* Technically, a MOVE uses UDUs for its units, but UDUs and GDUs are identical until a SHOW or WINDOW is executed.

Now you know what you're measuring—voltage vs. time—but the units are still not shown. As you saw in the last chapter, what is needed is an X-axis and a Y-axis, and they need to be labelled with numbers in appropriate places.

Miscellaneous Graphics Concepts

Clipping. Something that occurs completely "behind the scenes" in your computer when drawing is a process called clipping. Clipping is the process whereby lines that extend over the defined edges of the drawing surface are cut off at those edges. There are two different clipping boundaries at all times: the soft clip limits and the hard clip limits. The hard clip limits are the absolute boundaries of the plotting surface, and under no circumstances can the pen go outside these limits. The soft clip limits are user-definable limits, and are defined by the CLIP statement.

```
CLIP 10,20.5,Ymin,Ymax
```

This statement defines the soft clip boundaries only; hard clip limits are completely unaffected. After this statement has been executed, all lines which attempt to go outside the X limits (in UDUs) of 10 and 20.5, or the Y limits (in UDUs) of Ymin and Ymax will be truncated at the appropriate edge. Clipping *at the soft clip limits* can be turned off by the statement:

```
CLIP OFF
```

It can be turned back on, using the same limits, by

```
CLIP ON
```

If you want the soft clip limits to be somewhere else, use the CLIP statement with four different limits. Only one set of soft clip limits can be in effect at any one time. Clipping at the hard clip limits cannot be disabled.

The VIEWPORT statement, in addition to defining how WINDOW coordinates map into the VIEWPORT area, turns on clipping at the specified VIEWPORT edges.

Drawing Modes. On a monochromatic CRT, there are three different drawing modes available. (For selecting pens with a color CRT, see the section on color in this chapter.) The three pens perform the following actions:

Number	Function
1	Draws lines (turns on pixels).
−1	Erases lines (turns off pixels).
0	Complements lines (changes pixels' state).

A characteristic of drawing with pen −1 or pen 1 is that if a line crosses a previously drawn line, the intersection will be the same "color" as the lines themselves. When drawing with pen 0, and a line crosses a previously drawn line, the intersection becomes the opposite state of the lines. For example, assume a black background. You select PEN 0, then draw a pair of AXES. When the first axis is drawn, all pixels are off, so the line being drawn causes all pixels to be turned on along its length. However, when the second axis is drawn, it will turn on pixels until it gets to the other axis. At that point, the pixel is on, so it gets turned *off*. After that, the rest of the pixels are off, so they are again turned on.

Storing and Retrieving Images. If a picture on the screen takes a long time to draw, or the image is used often, it may be advisable to store the image itself—*not* the commands used to draw the image—in memory or on a file.

This may be done with the GSTORE command. First, you must have an INTEGER array of sufficient size to hold all the data in the graphics raster. This amounts to an array size of 7500 with Monochrome Plus black and white display, 12480 with Multimode black and white display, and 49920 with EGA color display. This array holds the picture itself, and it doesn't care how the information got to the screen, or in what order the different parts of the picture were produced. You can use GLOAD to restore the picture to the CRT display.

Data-Driven Plotting

When plotting data points, you will often find that they do not form a continuous line. You must have the ability to control the pen's position. In the last chapter, a passing reference was made to a third parameter in the PLOT statement. This third parameter is the pen-control parameter, and its function is to raise or lower the pen so that many lines can be drawn with one set of data.

When using a single X-position and Y-position in a PLOT statement (as opposed to plotting an entire array; we'll cover this a little later), the third parameter is defined in the following manner. Though it need not be of type INTEGER, its value should be an integer. If it is not, it will be rounded. The third parameter is either positive or negative, and at the same time, either even or odd. Whether the number is odd or even determines *which* action will be performed on the pen, and the sign of the number determines *when* that action will be performed: before or after the pen is moved.

Pen Control Parameter

	Even (Up)	**Odd (Down)**
Positive (after)	Pen up after move	Pen down after move
Negative (before)	Pen up before move	Pen down before move

The default parameter is +1—positive odd—therefore, the pen will drop after moving, and if the pen is already down, it will remain down, drawing a line. Zero is considered positive.

The program LEM1 (file "LEM1" on Manual Examples disc) is a good example of pen control. It draws a LEM (Lunar Excursion Module). There are two arrays used: a two-column REAL array for the X and Y data, and a one-column INTEGER array containing pen-control data. The data is read from DATA statements. Load and list the program and then run it.
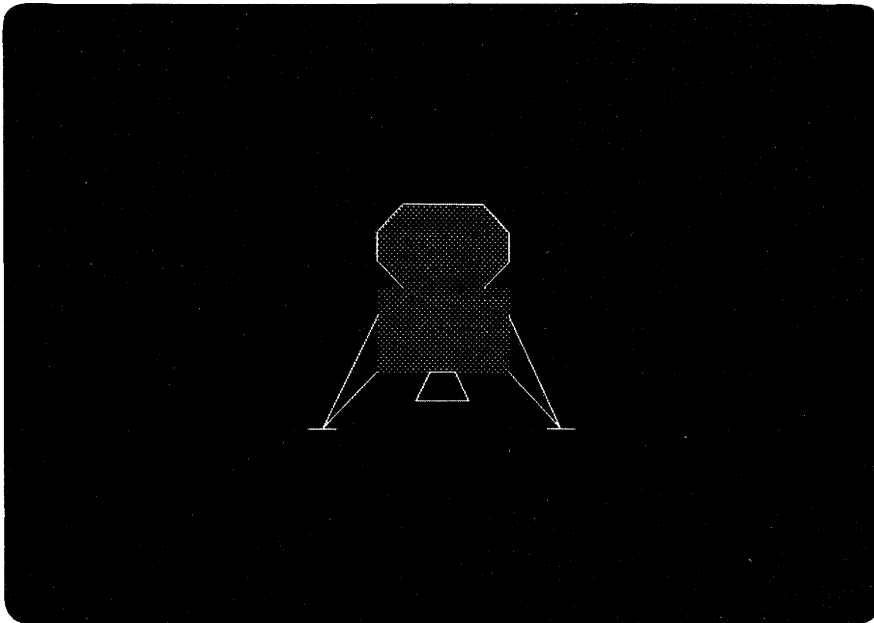
Having the pen-control parameter in a third column of the data array is generally a good strategy; it reduces the number of array names you must declare, and when you have the data points for the picture, you also have the information necessary to draw it. Nevertheless, an array must be entirely of one type, and usually you'll want the data to be real. So if you're pressed for memory, you may want to have a two-col-

umn data array of type REAL, and a one-column pen-control array of type INTEGER. Integer numbers take only one-fourth the memory real numbers take to store.

Translating and Rotating a Drawing

Often there is an application where a segment of a drawing must be replicated in many places; the same sub-picture needs to be drawn many times. Using the PLOT statement, it is possible but rather tedious to do. There is another statement called RPLOT, which draws a figure relative to a point of your choice. RPLOT means *Relative PLOT*, and it causes a figure to be drawn relative to a previously-chosen reference point. RPLOT's parameters may be two or three scalars, or a two-column or three-column array; the parameters are identical to those of PLOT.

The picture defined by the data given to an RPLOT statement is drawn relative to a point called the current relative origin. This is not necessarily the same as the pen position. The current relative origin is the last point resulting from any one of the following statements:

AXES	DRAW	FRAME
GINIT	GRID	IDRAW
IMOVE	IPLOT	LABEL
MOVE	PLOT	POLYGON
POLYLINE	RECTANGLE	

Typically, a MOVE is used to position the current relative origin at the desired location, then the RPLOT is executed to draw the figure. After the RPLOT statement has executed, the pen may be in a different place, but the current relative origin has not moved. Thus, executing two identical RPLOT statements, one immediately after the other, results in the figure being drawn precisely on top of itself.

A figure drawn with RPLOT can be rotated by using the
PIVOT statement before the RPLOT. PIVOT's single param-
eter is a numeric expression designating the angular distance
through which the figure is to be rotated when drawn. This
value is interpreted according to the current angular mode:
either DEG or RAD.

A program using RPLOT can be found on the Manual Exam-
ples disc under the file name "RPLOT". Load and list this file.
Various figures are defined with DATA statements: a desk, a
chair, a table, and a bookshelf. The program displays a floor
layout. Here again, the "end polygon mode" codes (the 0,0,7s
in the desk and chair definitions) are unnecessary; when a
polygon mode starts, any previous one ends by necessity.
Now run the program to see the figure below:
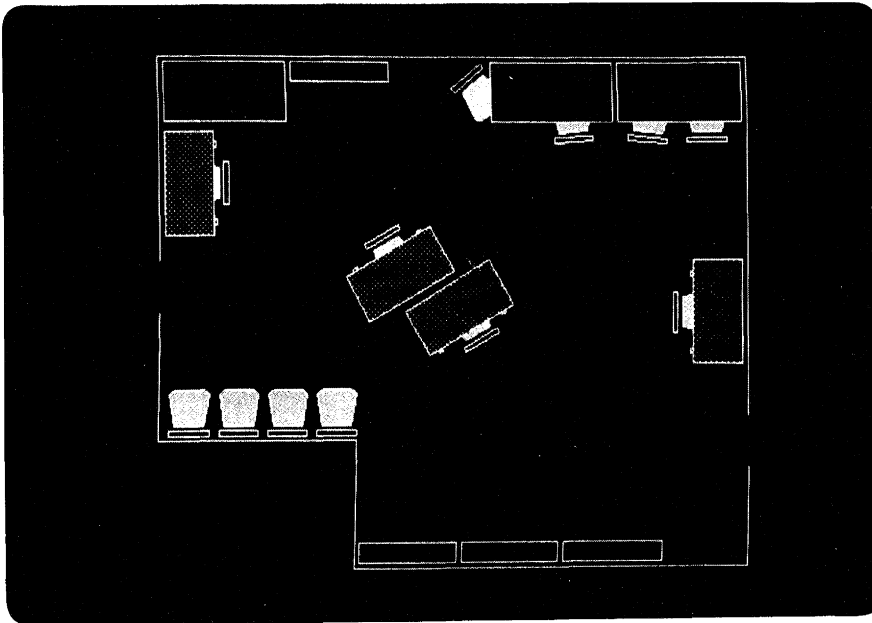
There are two points of interest in this program. First, notice that you can specify the EDGE and/or FILL parameters in the RPLOT statement itself (as in lines 230 and 260), or in the array (as in lines 180 and 210). FILLs and EDGEs are specified in the array by having a 6, a 10, or an 11 in the third column of the array. If FILL and/or EDGE are specified in both the PLOT statement and in the data, and the instructions differ, the value in the data replaces the FILL or EDGE keyword on the statement.
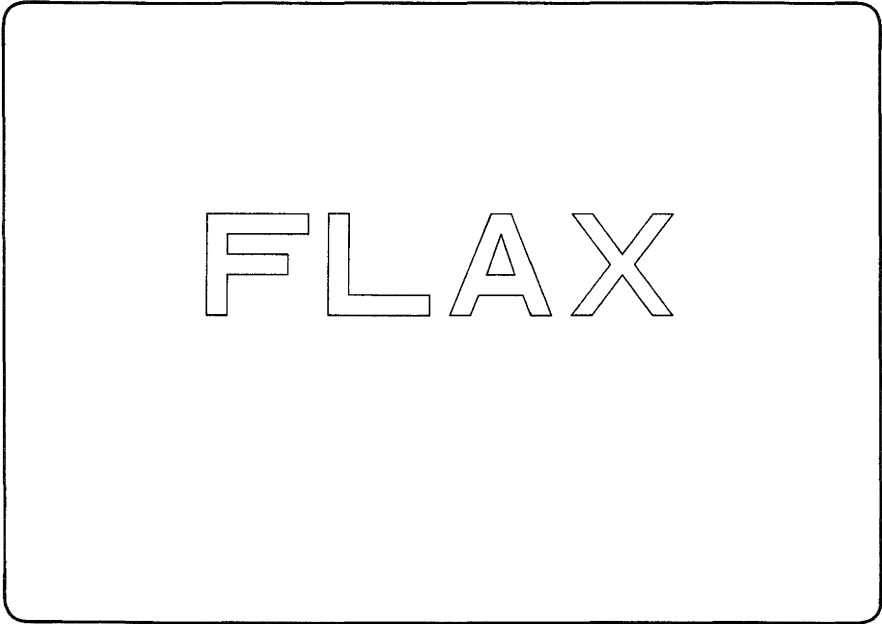
The second interesting point is that some of the chairs appear to be *under* the desks and tables; that is, parts of several chairs are hidden by other pieces of furniture. This is accomplished by drawing the chair, and then drawing the desk or table partially over the chair, and filling the desktop or tabletop with its own fill pattern, which may be black.

Incremental Plotting

Incremental plotting is similar to relative plotting, except that the origin, the point considered to be 0,0—is moved every point. Every time you move or draw to a point, the origin is immediately moved to the new point, so the next move or draw will be with respect to that new origin.

There are three incremental plotting statements available: IPLOT, which has the same parameters as PLOT and RPLOT; and IMOVE and IDRAW, which have the same parameters as MOVE and DRAW, respectively.

An example program using IPLOTs can be found in the file "FLAX" on the Manual Examples disc. It reads data from data statements describing the outlines of certain letters of the alphabet, and then plots them. Load and run this file.

A program which demonstates the use of POLYGON,
POLYLINE, PLOT, RPLOT, polygon filling, and gray-shading
can be found on the file "SCENERY" on the Manual Examples
disc. Load and run this program.

Points of note in this program:

- The sunrise was created with graduated gray shades in successively smaller "circles" (actually 30-sided polygons).
- The horizon was created by defining a rough edge on the top half of a polygon which blacked out the bottom section of the screen. This covered up the bottom of the sun. The white line of the horizon was simple plotting of the horizon array *without the first and last points*. We didn't want the lower corners of the screen to be included.
- The clouds were created by plotting "circles" after having invoked anisotropic units; thus long, thin ellipses resulted.

- The seagulls were created by drawing two arcs with POLYLINE. An arc is created by defining an N-sided polygon and drawing less-than-N sides. Note that PIVOT was used to cause the starting angle of the arcs to be other than straight to the right.
- The trees were created by defining an array whose left side is a mirror image of the right side. The array is centered around zero in the X direction to allow for scaling of the tree simply by multiplying the array by a constant. RPLOT was used to place the trees in their various positions.

External Graphics Displays and Plotters

Specifying a Plotter

In previous sections you saw program listings containing a line with a PLOTTER IS statement:

```
PLOTTER IS 3,"INTERNAL"
```

This caused the computer to activate the internal CRT graphics raster as the plotting device, and thus all subsequent commands were directed to the screen. If you want a plotter to be the output device, only the PLOTTER IS statement needs to be changed. If your plotter is at interface select code 7 and address 5 (the factory settings), the modified statement would be:

```
PLOTTER IS 705,"HPGL"
```

"HPGL" stands for Hewlett-Packard Graphics Language, and it is the low-level language which the plotters actually speak behind the scenes. More about this later.

There are some limitations, though. If you are doing an operation on one plotting device and attempt to send the plot to another device which does not support that operation, it won't work.

For example: area fills, which are valid operations on the internal CRT, are not available on plotters. Color map operations, which are valid on the internal CRT (of the high resolution color display), are not valid on a plotter. Erasing lines can be done on the internal CRT and the external monitors, but, naturally, not on a hard-copy plotter. HPGL commands will be interpreted correctly by a hard-copy plotter, but not by the internal CRT.

Using a Shared Printer or Plotter

Use of special Shared Resource Manager (SRM) directories called spooler directories allows you to access a shared plotter. Setting up a spooler directory is explained in the "SRM Server Startup" chapter of the *SRM Software Installation* manual. The following examples assume that the spooler directory PL has been created at the root of the SRM directory structure.

```
MSI ":REMOTE"
```

Include in your plotting program:

```
CREATE BDAT "PL/plot-file",1
PLOTTER IS "PL/plot-file"
   .
   .
   .
PLOTTER IS 3,"INTERNAL"
```

The PLOTTER IS statement only works with BDAT files.

Dumping Raster Images

In addition to generating a hard-copy plot with a plotter, as described above, you can dump a CRT's raster image to a printer. This method is called a *graphics dump* or *screen dump*. It is accomplished by copying data from the frame buffer to a printer to be printed dot for dot.

First, the image must be drawn on a CRT. Since this technique dumps a raster-type image, it prints only dots. Thus, it cannot draw a line, per se, but only the approximation of a line from the screen, made up of dots. The dump device "takes a snapshot" of the graphics screen at some point in time, and doesn't care *how* the dots came to be turned on or off. Thus, filled areas can be dumped to the printer; indeed, all CRT graphics capabilities (except color) are available.

If your printer is an HP 9876, HP 2631G, HP 2671G, HP 2673A, HP Laser Jet, HP ThinkJet, or any other printer which conforms to the HP Raster Interface Standard, dumping graphics images is easy. For example:

```
100 DUMP DEVICE IS 26
110 DUMP GRAPHICS
```

or simply,

```
100 DUMP GRAPHICS #26
```

Both of these program segments would take the image in the last specified CRT graphics frame buffer (the internal CRT by default) and send it to the printer at address 26. If no device is specified, the image is taken from the last active CRT, whether internal or external. The default factory setting for printers is 701. You would probably use the two-statement version in an application where you wish to specify the destination device once, and have it apply to many different DUMP GRAPHICS statements. The one-statement version would probably be used where there are few and isolated DUMP GRAPHICS statements.

DUMP GRAPHICS will also send a graphics display to a printer. If a DUMP DEVICE IS statement has not been executed, the dump device is expected to be at address 701.

If a DUMP GRAPHICS operation is aborted with CLR I/O, the printer may or may not terminate its graphics mode. Sending 75 null characters (ASCII code zero) to a printer such as a HP 9876 terminates its graphics mode. For example:

```
OUTPUT Dump_dev USING "#,K";RPT$(CHR$(0),75)
```

If you want the image to be twice as large in each dimension as the actual screen size, you can specify:

```
100 DUMP DEVICE IS 701,EXPANDED
110 DUMP GRAPHICS
```

This will cause the dumped image to be four times larger than it would be if EXPANDED had not been specified. Each dot is represented by a 2 × 2 square of dots, and the resulting image is rotated 90° clockwise to allow more of the resulting image to fit on the page.

If you have a printer which does not conform to the HP Raster Interface Standard, all is not lost. It must, however, be capable of printing raster-image bit patterns.

HPGL Hewlett-Packard Graphics Language (HPGL) is a low-level language that is understood by all current HP hard-copy plotters. When you specify:

```
PLOTTER IS 705,"HPGL"
```

the plotter specifier "HPGL" notifies the computer that it will be talking with a device which understands HPGL. This causes all the user's BASIC statements to be converted into HPGL commands and sent to the plotter. HP plotters always receive commands in HPGL.

When you are executing BASIC graphics statements and they are doing operations on an HP plotter, there is nothing preventing you from interspersing your own HPGL commands between the BASIC commands. HPGL commands can be sent to the device with PRINT statements, after having specified the receiving device in a PRINTER IS statement, but the preferred way is to use the OUTPUT statement. HPGL command sequences are terminated by a linefeed, a semicolon, or an EOI character, which is sent by the HP-IB (Hewlett-Packard Interface Bus) END keyword. Individual commands within a sequence are typically delimited by semicolons.

There are many HPGL commands available, but the exact ones you will be able to use depend on the device itself. Plotters are not the only devices which use HPGL; digitizers and graphics tablets do also. By their nature, however, they use a different subset of commands than plotters do. Following are a few of the more common and/or useful HPGL commands.

Controlling Pen Speed. If your plotter pens are getting old, you probably would want to make them draw more slowly to get a better quality line. (There are other factors which can affect line quality. For example, humidity can alter the line quality of a fiber-tipped pen.) To accomplish this, you could have a statement:

```
OUTPUT 705;"VS10"
```

"VS" stands for "Velocity Select" and the "10" specifies centimeters per second. Thus, this statement would tell the plotter to draw at a maximum speed of ten centimeters per second. It specifies a *maximum* speed rather than an *only* speed, because on short line segments, the pen does not have time to accelerate to the specified speed before the midpoint of the line segment is reached and deceleration must begin. The range and resolution of pen speeds, and default maximum speed depend on the plotter.

Controlling Pen Force. On the HP 7580 and HP 7585 drafting plotters, you can specify the amount of force pressing the pen tip to the drawing medium. This is useful when matching a pen type (ball-point, fiber-tip, drafting pens, etc.) to a drawing medium (paper, vellum, or mylar, etc.). Again, if a pen is partially dried out, it may help line quality to adjust the pen force.

An example statement is:

```
OUTPUT 705;"FS3,6;"
```

This statement (*Force Select*) would specify that pen number 6 should be pressed onto the drawing medium with force number 3. As you can see, the force specifier occurs first, the pen number second. The reason for this is that if you do *not* specify a pen number, all pens will be affected.

The force number is translated into a force in grams. If, for example, you have an HP 7580A plotter, the force number is converted to force as follows:

1 = 10 grams   4 = 34 grams   7 = 58 grams

2 = 18 grams   5 = 42 grams   8 = 66 grams

3 = 26 grams   6 = 50 grams

Selecting Character Sets. Some plotters contains internal character sets which may be much more pleasing to the eye or more appropriate for your application than the character set provided by the BASIC operating system. Through HPGL, you can tell the plotter to use these character sets.

```
OUTPUT 705;"CS1"
```

tells the plotter to use character set 1 until further notice. This means, however, that to actually get these characters, you cannot use the LABEL statement in BASIC. This is because the BASIC graphics system generates all its characters as a series of line segments, and the plotter can't tell when it is told to draw a line segment whether it is going to be part of a character or not. Thus, you must use the HPGL label command, LB:

```
OUTPUT 705;"LBThis is an example string."&CHR$(3)
```

CHR$(3) is the End-of-text or ETX character. It is the default terminator for the LB command. If you wish, you can specify other characters to signal the end of a line of text to label. You use the *Define Terminator* command:

```
OUTPUT 705;"DT&"
```

This statement instructs the plotter to consider the ampersand to be the terminator. Thus, every LB command must have an ampersand as the final character.

Note 👆 When using a printable ASCII character as the terminator, *it will be labelled* in addition to terminating the LB command. Also, there must be a terminator as the final character in the string to indicate the end of the text, or all subsequent commands will be considered text and not commands; that is, they will merely be labelled, not executed.

Error Detection. When using HPGL commands, there is always a possibility of making an error. When this occurs, the program should be able to respond in a friendly way, and not just hang then and there. With HPGL, it is possible to interrogate the plotting device and determine the problem. The following statements in an error-trapping routine would determine the type of error that occurred:

```
OUTPUT 705;"OE;"
ENTER 705;Error
```

After these two statements have executed, the variable Error will contain the number of the *most recent* error. What the error code means depends on the particular device being used.

This is not by any means an exhaustive list of HPGL commands, but it serves to acquaint you with the concept of using the HPGL language, and the amount of control it gives you over the peripheral device. A thorough understanding of HPGL can only be gotten by combining information from the owner's manual of the particular device you have with actual hands-on experience.

Color Graphics

Color can be used for emphasis, clarity, and to present visually pleasing images. Color is a very powerful tool, and it follows directly that it is very easy to misuse. Be careful in using color, and it will serve as a valuable tool for communication. Misuse it, and it will garble the communiation.

The biggest benefit of the color computer is that it makes experimenting with color so easy. With a bit-mapped frame buffer and a color map, it is easy to test out ideas before you use them. It is also possible to use the color map for simple animation effects and some just plain impressive images.

The methods for displaying color fall into four categories:

- *Background Value.* Whenever GCLEAR is executed, all the pixel locations in the display are set to 0. Thus, PEN 0 is the background color.
- *Line Value.* The PEN statement is used to determine the color written to the display for all lines drawn. This includes all lines (including characters created by LABEL) and outlines specified by the secondary keyword EDGE.
- *Fill Value.* The AREA PEN statement is used to specify the color written to the display for filling areas specified by the secondary keyword FILL.
- *Dithered Colors.* AREA INTENSITY and AREA COLOR can also be used to specify a fill color.

The PEN, AREA PEN, AREA INTENSITY, and AREA COLOR statements control what are referred to as *modal attributes*. This means that the value established by one of the statements stays in effect until it is altered by another statement.

Non-Color Mapped Color

When PLOTTER IS CRT, "INTERNAL" is executed, 8 colors are available through the PEN and AREA PEN statements. The colors provided are:

- Black and white.

- Red, green, and blue (the additive color primaries).
- Cyan, magenta, and yellow (the complements of the additive color primaries).

The colors can be selected with the PEN statement, the same way they are for an external plotter. The colors and their pen-selectors are listed below.

Non-Color-Mapped Pens. The meanings of the different pen values are shown in the table below. The pen value can cause either a 1 (draw), a 0 (erase), n/c (no change), or complement (invert) the value in each color plane.

Non-Color Map Mode

Pen	Action	Plane 1 (red)	Plane 2 (green)	Plane 3 (blue)
−7	Erase Magenta	0	n/c	0
−6	Erase Blue	n/c	n/c	0
−5	Erase Cyan	n/c	0	0
−4	Erase Green	n/c	0	n/c
−3	Erase Yellow	0	0	n/c
−2	Erase Red	0	n/c	n/c
−1	Erase White	0	0	0
0	Complement	invert	invert	invert
1	Draw White	1	1	1
2	Draw Red	1	0	0
3	Draw Yellow	1	1	0
4	Draw Green	0	1	0
5	Draw Cyan	0	1	1
6	Draw Blue	0	0	1
7	Draw Magenta	1	0	1

If you are in this mode, you can draw lines in the eight colors listed above. The following program (found in file COLORLINE on your Manual Examples disc) shows the colors available.

```
10  GINIT
20  GRAPHICS ON
30  MOVE 20,80
40  FOR X=1 TO 7
50    PEN X
60    IDRAW 50,0
70    IMOVE -50,-10
80  NEXT X
90  END
```

Color Mapped Color

If you are trying to define a complex human interface, you will need more colors and more control over the colors. This is possible after you turn on the color map. To do so, execute:

```
PLOTTER IS CRT,"INTERNAL";COLOR MAP
```

Default Colors. If you do not modify the color map, the colors selected by the PEN and AREA PEN values depend on the default color map values. These values are shown in the following table:

Default Color Map and Pen Values

Pen	Color
0	Black
1	White
2	Red
3	Yellow
4	Green
5	Cyan
6	Blue
7	Magenta
8	Black
9	Olive Green
10	Aqua
11	Royal Blue
12	Maroon
13	Brick Red
14	Orange
15	Brown

The colors of pens 9–15 are only available in the separate alpha/graphics mode. In the combined alpha/graphics mode (the default mode), pens 8–15 are mapped to the colors of pens 0–7.

Pens 0–7 of the default color map are the same as in non-color map mode. The upper 8 colors (8 through 15) were selected by a graphic designer to produce graphs and charts for business applications. The colors are:

■ Maroon, Brick Red, Orange, and Brown (warm colors).
■ Black, Olive Green, Aqua, Royal Blue (cool colors).

These colors are one designer's idea of appropriate colors for business charts and graphs. They were chosen to avoid clashing with each other.

Changing Default Colors. The SET PEN statement is used to customize the color that each PEN value represents. SET PEN supports two color models, the RGB (Red, Green, Blue) model and the HSL (Hue, Saturation, Luminosity) model. Since the color models are dynamically interactive, it is much easier to understand them by experimenting with them.

You can think of the RGB model as mixing the output of three light sources (one each for red, green, and blue). The parameters in the model specify the intensity of each of the light sources. The RGB model is accessed through the secondary keyword INTENSITY used with the SET PEN statements. The values are *normalized* (range from 0 through 1). Thus,

```
SET PEN 0 INTENSITY 0.7, 0.7, 0.7
```

sets pen 0 (the background color) to approximately a 70% gray value. Whenever all the guns are set to the same intensity, a gray value is obtained. The parameters for the INTENSITY mode of SET PEN are in the same order they appear in the name of the model, red, green, and blue.

When using an EGA system, each primary color (red, green, and blue) can be displayed at four distinct levels:

- off
- ⅓ on
- ⅔ on
- full on

Therefore, each PEN may be set to one of 64 distinct colors.

The HSL model is closer to the intuitive model of color used by artists, and is very effective for interactive color selection. The three parameters represent *hue* (the pure color to be worked with), *saturation* (the ratio of the pure color mixed with white), and *luminosity* (the brightness-per-unit area). The HSL model is accessed through the SET PEN statement with the secondary keyword COLOR:

```
SET PEN Current_Pen COLOR Hue, Saturation, Luminosity
```

Hue, Saturation, and Luminosity are normalized to values from 0 to 1.

Fill Colors

In either color-mapped or non-color-mapped mode, areas may be filled with a PEN color by first selecting that PEN with an AREA PEN statement. Filling is specified by using the secondary keyword FILL in any of the following statements:

IPLOT   PLOT   POLYGON

RECTANGLE   RPLOT   SYMBOL

It is possible to fill areas with other shades. These tones are achieved through *dithering*. Dithering produces different shades by combining dots of the eight colors described earlier. The screen is divided into 4-by-4 cells, and patterns of dots within the cells are turned on to match, as closely as possible, the color you specify. Dithered colors are defined with the AREA COLOR and AREA INTENSITY statements using the RGB or HSL models described in the previous section.

5 Interface Techniques

Contents

Chapter 5

5

Interface Techniques

This chapter describes the functions and requirements of interfaces between your computer and its resources. You can gain useful information that will increase your understanding of interfacing equipment with the computer. More detailed information can be found in the BASIC 5.0 Interfacing Techniques (two volumes).

Terminology

The terms used in this chapter are important to your understanding of the information presented here.

The term "computer" is defined as the processor, its support hardware, the MS-DOS operating system, and the BASIC language system. Together, these system elements manage all computer resources. The term "computer resource" is used to describe all of the "data handling" elements of the system. Computer resources include internal memory, CRT display, keyboard, disc drive, and any external devices that are under computer control (printer, plotter, etc.). These resources are often referred to as "peripheral" devices.

The term **hardware** describes both the electrical connections and the electronic devices that make up the circuits within the computer. Any piece of hardware is an actual physical device. The term **software** describes the user-written, BASIC language programs. **Firmware** refers to the pre-programmed assembly language programs that are invoked by BASIC language statements and commands, or assembly language routines of the operating system. You cannot modify firmware.
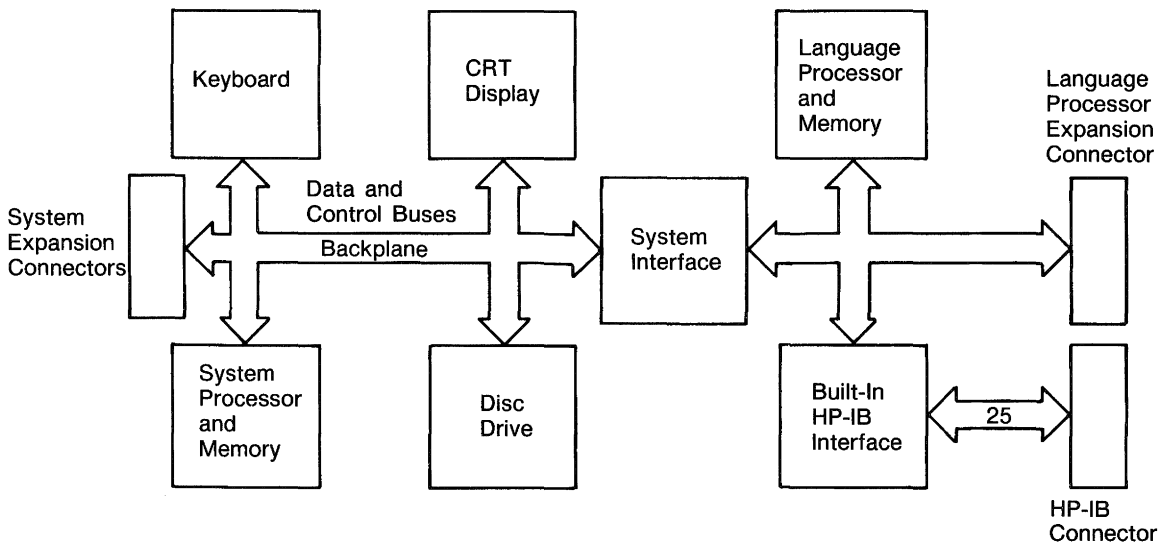
The term **I/O** is an acronym for "Input and Output". It refers to the process of copying data to or from the computer's memory.

The term **bus** refers to a common group of hardware lines that are used to transmit information between computer resources. The computer communicates directly with the internal resources through the data and control busses. The **computer backplane** is an extension of these internal data and control busses. The computer communicates indirectly with the external devices through **interfaces** connected to the backplane.

Why Do You Need an Interface?

The functions of an interface are shown in the following block diagram:
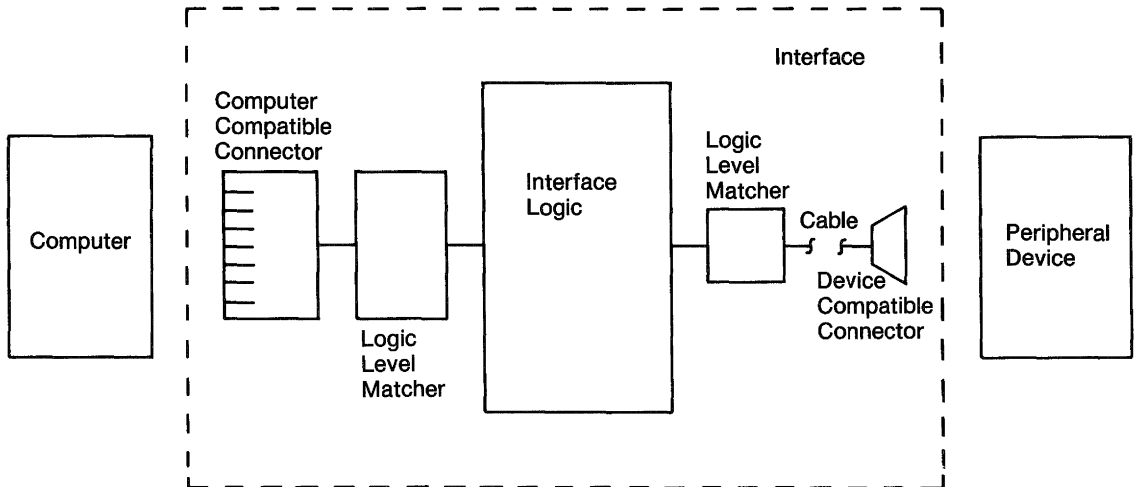
The primary function of an interface is to provide a communication path for data and commands between the computer and its resources.

The following explains the need for interfaces.

Electrical and Mechanical Compatibility

Electrical compatibility must be insured before any thought of connecting two devices occurs. The two devices often have input and output signals that do not match. If so, the interface serves to match the electrical levels of these signals before the physical connections are made.

Mechanical compatibility simply means that the connector plugs must fit together properly. Most interfaces have cables available that can be connected directly to the device.

Data Compatibility

The computer and the peripheral device must agree upon the form and meaning of data before communicating it. Some interfaces format data, but most have little responsibility for matching data formats. The computer must generally make the necessary changes, if any, so that the receiving device gets meaningful information.

Timing Compatibility

Since all devices do not have standard data transfer rates and do not always agree as to when the transfer will take place, a consensus between the sending and receiving devices must be made.

If the data transfer is not begun at an agreed upon point in time and at a known rate, the transfer must proceed one data item at a time with acknowledgement from the receiving device that it has the data and that the sender can send the next item. This process is known as a "handshake".

Additional Interface Functions

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

Interface Overview

Each of the interfaces discussed in the following section is designed for a specific method of data transfer.

The HP-IB Interface

The HP-IB interface is Hewlett-Packard's implementation of the IEEE-488 1978 Standard Digital Interface for Programmable Instrumentation. The acronym HP-IB stands for "Hewlett-Packard Interface Bus", and is often referred to as the "bus."
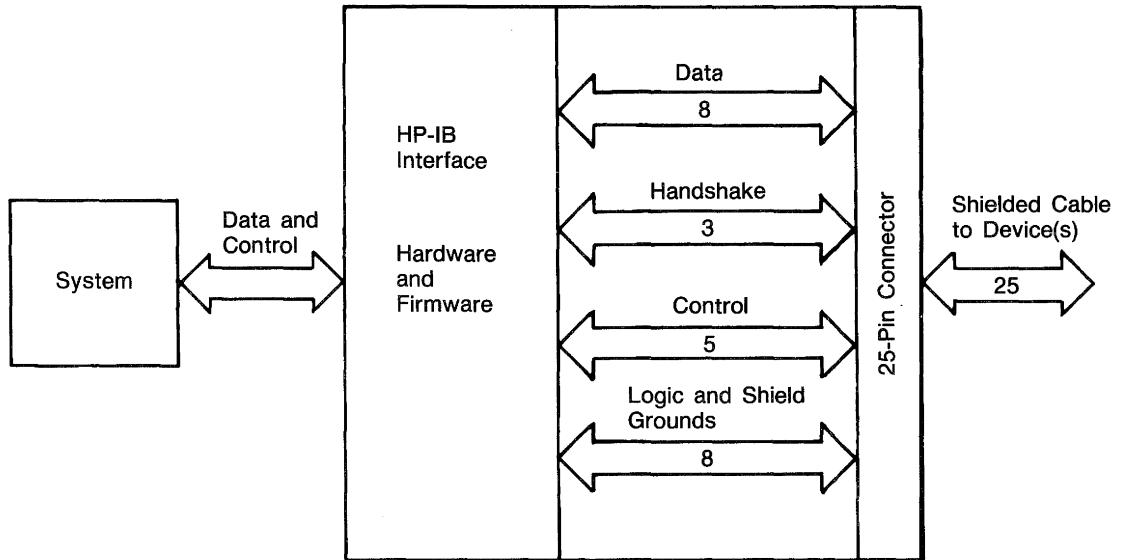
The HP-IB interface fulfills all four compatibility requirements (hardware, electrical, data, and timing) with no additional modification. All you need to do is connect the interface cable to the desired HP-IB.

The "bus" is somewhat of an independent entity. It is a communication arbitrator that provides an organized protocol for communications between several devices. The bus can be configured several ways. The devices on the bus can be configured as senders or receivers of data and control messages, depending on their capabilities.

The RS-232 Serial Interface

The serial interface changes 8-bit parallel data into 8-bit-serial information and transmits the data through a two-wire cable. Data is received in this serial format and is then converted back to parallel data. This use of two-wire cable makes it more economical to transmit data over long distances than would be the case if eight individual lines were used.
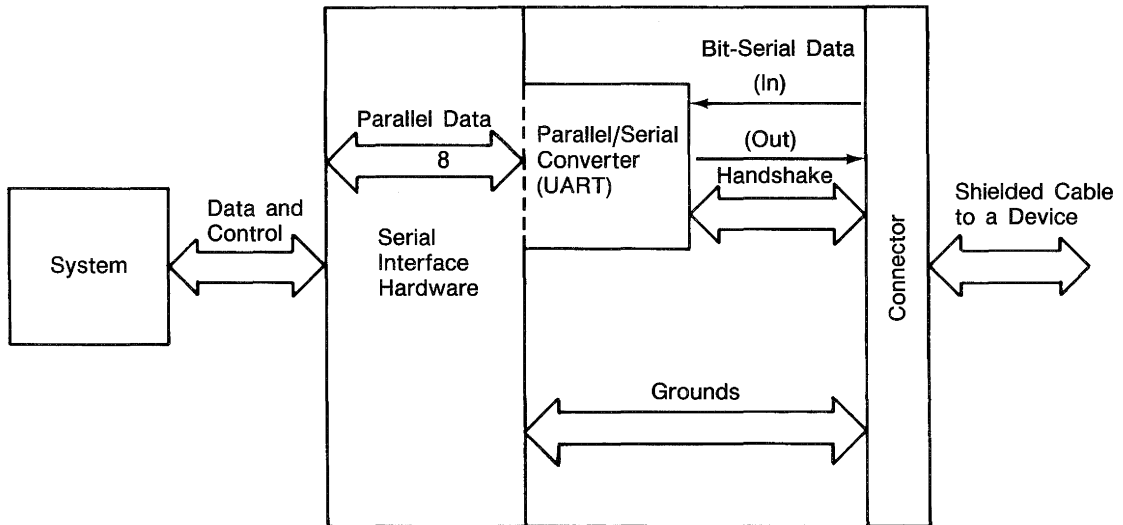
Data is transmitted at several programmable rates using either a simple data handshake or no handshake at all. The main use of this interface is communicating with simple devices.

The GPIO Interface

The GPIO (General Purpose Input/Output) interface provides the most flexibility of all the interfaces. It consists of 16 output data lines, 16 input data lines, two handshake lines, and other assorted control lines. Data is transmitted using programmable handshake conventions and logic sense.

Parallel Data Out
16

Parallel Data In
16

Handshake
4

Special Purpose
6

Grounds
7

Backplane Connector

Data and Control
100

GPIO Interface Hardware

50-Pin Connector

Shielded Cable to a Device
50
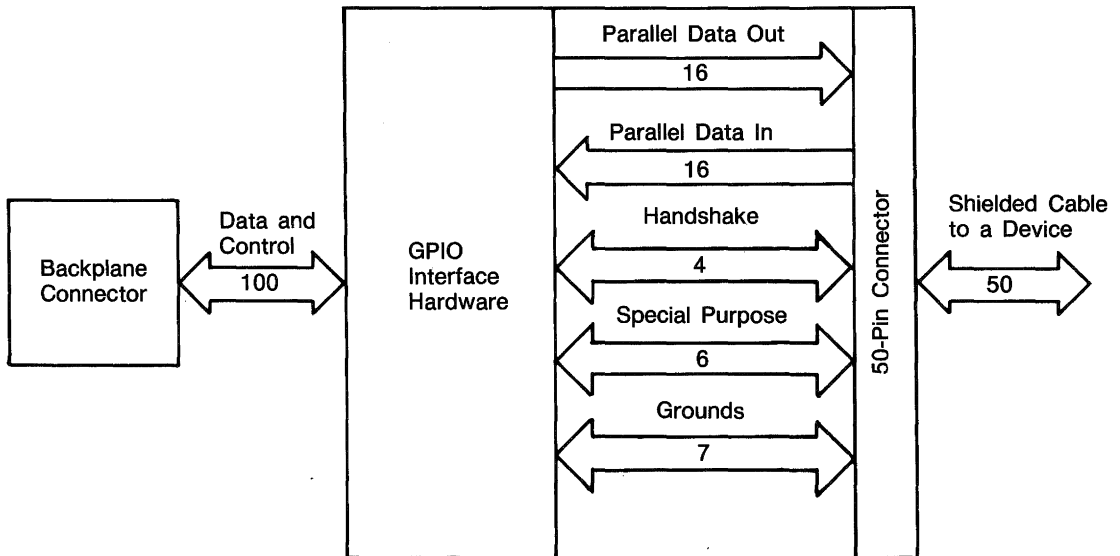
The I/O Process

The I/O process begins when the computer encounters an I/O statement in a program. The computer first determines the type of I/O statement to be executed (such as ENTER US-ING, OUTPUT, etc.). Once the type of statement is determined, the computer evaluates the statement's parameters.

Specifying a Resource

Each resource must have a unique specifier that allows it to be accessed to the exclusion of all other resources connected to the computer. The methods of uniquely specifying re-sources are device selectors, string variable names, and path names.

For example, before executing an OUTPUT statement, the computer first evaluates the parameter which specifies the destination resource. The source parameter of an ENTER statement is evaluated in the same manner.

```
OUTPUT Dest_parameter;Source_item

ENTER Source_parameter;Dest_item
```

Registers

The computer must often read certain memory locations to determine which firmware routines will be called to execute the I/O procedure. The contents of these locations, known as registers, store parameters to be used and the type of interface involved in the operation.

An example of register usage by firmware is during output to the CRT. Characters output to this device are displayed beginning at the current screen coordinates. After the computer has evaluated the first expression in the source-item list, it must determine where to begin displaying data on the screen. Two memory locations are dedicated to storing the "X" and "Y" screen coordinates. The firmware determines these coordinates and begins copying the data to the corresponding locations in display memory.

Data Handshake

Each byte (or word) of data is transferred with a procedure known as data-transfer handshake. It is the means of moving one byte of data at a time when the two devices are not in agreement as to the rate of data transfer or as to what point in time the transfer will begin. The steps of the handshake are as follows:

1. The sender signals to get the receiver's attention.

2. The receiver acknowledges that it is ready.

3. A data byte (or word) is placed on the data bus.

4. The receiver acknowledges that it has received the data item and is now busy. No further data may be sent until the receiver is ready.

5. Repeat these steps if more data is to be transferred.

I/O Examples

Now that you have seen the steps taken by the computer when executing an I/O statement, lets look at how two typical I/O statements are executed by the computer.

Example OUTPUT Statement

Data can be output to only one resource at a time with the OUTPUT statement (with the exception of the HP-IB interface). This destination can be any computer resource, and is specified by the destination parameter as shown below:

```
OUTPUT Destination; String$,CHR$(C+32),"That's all"
```

The data to be output may be a string, the result of a function, or a constant. Either string or numeric expressions can specify the actual data to be output.

Sample ENTER Statement

Data can be entered from only one resource at a time. The source can be any resource, and is specified by the source parameter as shown by the following statement:

```
ENTER Source;Number,String$
```

The destinations of ENTER operations are always variables in memory. Both string and numeric variables can be specified as the destinations.

Directing Data Flow

As described in the previous section, data can be moved between computer memory and several resources, including:

- Computer memory (string variables in memory).
- Internal and external devices.
- Mass storage files.
- Buffers.

This section describes how string variables and devices are specified in I/O statements.

Specifying a Resource

Each resource must have a specifier that allows it to be accessed to the exclusion of all other resources. String variables are specified with their names, while devices can be specified with either their device selector or with a new data type known as an I/O path name. This section describes how to specify these resources in OUTPUT or ENTER statements.

String-Variable Names. Data is moved to and from string variables by specifying the variable's name in an OUTPUT or ENTER statement. Examples of each are shown in the following program (found in file OUTENTER on your Manual Examples disc):

```
100   DIM To_dest$[80],From_source$[80]
110   DIM Data_out$[80]
120   !
130   From_source$="Source data"
140   Data_out$="OUTPUT data"
150   !
160   PRINTER IS 1
170   PRINT "To_dest$ before OUTPUT= ";To_dest$
180   PRINT
190   !
200   OUTPUT To_dest$;Data_out$;
210   PRINT "To_dest$ after OUTPUT= ";To_dest$
220   PRINT
230   !
240   ENTER From_source$;To_dest$
250   PRINT "To_dest$ after ENTER= ";To_dest$
260   PRINT
270   !
280   END
```

Printed results from the program are:

```
To_dest$ before OUTPUT=
To_dest$ after OUTPUT= OUTPUT data

To_dest$ after ENTER= Source data
```

Device Selectors. Devices include the built-in CRT and keyboard, plus external printers and instruments, and all other physical entities that can be connected to the computer through an interface. Each interface has a unique number by which it is identified, known as its interface select code. The internal devices are accessed with the following permanently assigned interface select codes:

Crt Display   1

Keyboard   2

Built-in HP-IB 7

Optional HP-IB and serial interfaces have select codes that you can set by means of the configuration file (refer to appendix F).

Other optional interfaces have select codes that you can set by means of switches on the interface card. These interfaces cannot use select codes 1 through 7; the valid range is 8 through 31. The following settings on optional interfaces have been made at the factory, but can be reset to any unique select code between 8 and 31. Refer to the interface instruction manual for further information.

GPIO   12

SRM   21

Examples of using interface select codes to access devices are shown below.

```
OUTPUT 1;"Data to CRT"
ENTER 1;Crt_line$

Int_sel_code=12
OUTPUT Int_sel_code;String$&"Expression",Num_expression
ENTER Int_select_code;Str_variable$,Num_variable

Number=2
ENTER Number+7;Serial_data$
OUTPUT 11-Number;"Data to serial card"
```

The device selector can be any numeric expression that rounds to an integer in the range 1 through 32 (32 is a pseudo select code used as a device selector for parity, cache, and float registers). If the interface select code specifies an HP-IB interface, additional information must be specified to access a particular HP-IB device, since more than one device can be connected to the computer through HP-IB interfaces.

HP-IB Device Selectors. Each device on the HP-IB interface has a primary address by which it is uniquely identified. Each address must be unique so that only one device is accessed when one address is specified. The device selector is therefore a combination of the interface select code and the devices address. Two examples are shown below.

To access the device on:

Interface select code 7 at primary address 01, use device selector 701.

Interface select code 10 at primary address 13, use device selector 1013.

I/O Path Names. All data entered into and output from the computer is moved through the "I/O path". An I/O path consists of the hardware and operating system firmware used to carry out this moving process. When a string variable or device selector is specified in an ENTER or OUTPUT statement, the operating system first evaluates the expression that specifies a resource, and then chooses the corresponding default I/O path through which data will be moved.

The I/O paths to devices and mass storage files can be assigned special names; I/O paths to string variables can only be assigned names if the variable is declared as a buffer. Assigning names to I/O paths provides improvements in performance and additional capabilities over using device selectors.

Assigning I/O Path Names

An I/O path name is a new data type that can be assigned to either a device or a data file on a mass storage device. Any valid name preceded by the "@" character can be used.

Note

A name is a combination of 1 to 15 characters, beginning with an upper case alphabetic character or one of the characters CHR$(161) through CHR$(254) and followed by up to 14 lower case alphanumeric characters, the underscore character (_), or the characters CHR$(161) through CHR$(254).

The following examples show you how this is done.

```
ASSIGN @Display 1
ASSIGN @Printer 26
ASSIGN @Serial 9
ASSIGN @Gpio 12
```

Now you could use the I/O path names instead of the device selectors to specify the resource with which the communication is to take place.

```
OUTPUT @Display;"Display message"
OUTPUT @Printer;"Message to the printer"
ENTER @Serial;Variable,Variable$
ENTER @Gpio;Word1,Word2
```

Since an I/O path name is a data type, a fixed amount of memory is allocated for the variable, similar to the manner in which memory is allocated to other program variables (integer, real, and string).

Attempting to use an I/O path name that does not appear in any program line results in error 910 ("Identifier not found in this context"). This error message indicates that memory space has not been allocated for the variable. Attempting to use an I/O path name that does appear in an ASSIGN statement in the program, but which has not yet been executed results in error 177 ("Undefined I/O path name"). This error indicates that memory space has been allocated, but no valid information has been placed into the variable since the I/O path name has not yet been assigned to a resource.

Reassigning I/O Path Names. If an I/O path name already assigned to a resource is to be reassigned to another resource, the preceding form of the ASSIGN statement is used. The first action is that the I/O path name to the device is implicitly closed. A new assignment is then made as though the first never existed.

```
100   ASSIGN @Printer TO 1 !Initial assignment.
110   OUTPUT @Printer;"Data1"
120   !
130   ASSIGN @Printer TO 701 !2nd ASSIGN closes the 1st
140   OUTPUT @Printer;"Data2" !and makes a new assignment.
150   PAUSE
160   END
```

The result of running the program is that "Data1" is sent to the CRT, and "Data2" is sent to the HP-IB device 701. Since the program was paused (which maintains the program context), the I/O path name @Printer can be used in an I/O statement or reassigned to another resource from the keyboard.

Closing I/O path names. A second use of the ASSIGN statement is to explicitly close the name assigned to an I/O path. Examples of statements that close path names are as follows.

```
ASSIGN @Printer TO *
ASSIGN @Serial TO *
ASSIGN @Gpio TO *
```

After executing this statement for a particular I/O path name, the name cannot be used in subsequent I/O statements until it is reassigned.

The HP-IB
Interface

This section describes the techniques necessary for programming the HP-IB interface. It also describes the specific details of how this interface works and how it is used to communicate with and control systems consisting of various HP-IB devices. Be sure you have the TRANS and IO binaries loaded in your system.

The HP-IB (Hewlett-Packard Interface Bus), commonly called the "bus", provides compatibility between the computer and external devices conforming to the IEEE 488-1978 standard. Electrical, mechanical, and timing compatibility requirements are all satisfied by this interface. The following diagram depicts the HP-IB interface.*
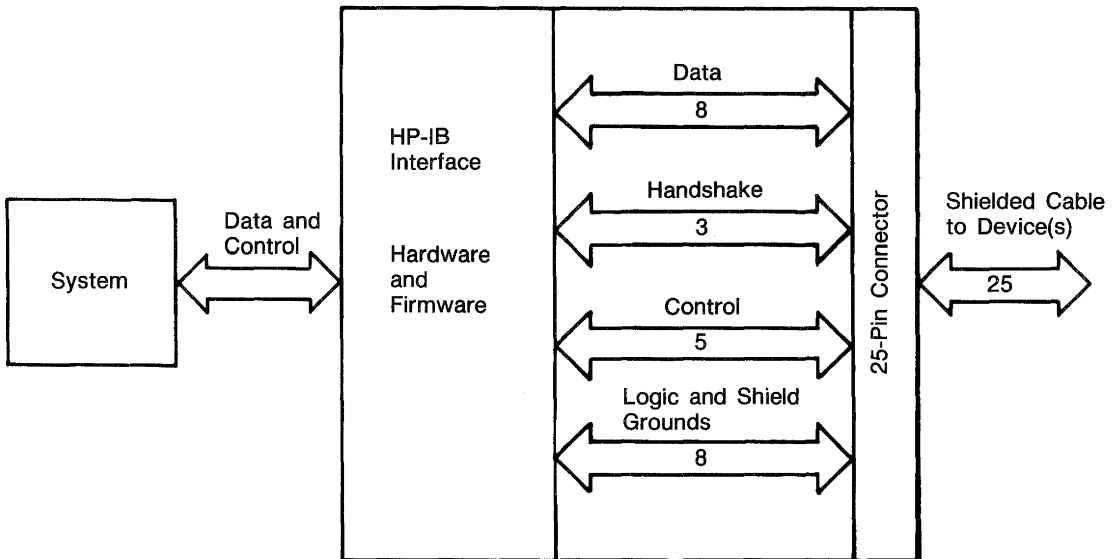
The HP-IB Interface is easy to use and allows great flexibility in communicating data and control information between the computer and external devices.

* See HP-IB Standard for listing of all 24 lines (cable printouts).

Initial Installation

The built-in HP-IB interface requires no installation. Refer to the HP-IB Interface manual and appendix F for information about setting the switches and installing an external HP-IB interface. Once the interface has been properly installed, you can verify that the settings are what you intended by running the program DEFAULT1 on your Manual Examples disc. You can also check the defaults of the internal HP-IB interface with the program. The results are displayed on the CRT.

The hardware interrupt level on the Language Processor card is set to 3 for the internal HP-IB interface, but can range from 3 to 6 on external interfaces. Refer to appendix F for information on mapping these levels to your computer's interrupt levels. Primary address is further described in "HP-IB Device Selectors" in the next section.

The term "System Controller" is also further described later in this section in "General Structure of the HP-IB". The internal HP-IB has a jumper that is set at the factory to make it a system controller. Refer to the language processor card installation guide for a description of how to change this jumper. External HP-IB interfaces have a switch that controls this interface state.

Communicating with Devices

This section describes programming techniques used to output data to and enter data from HP-IB devices. General bus operation is also briefly described in this section.

HP-IB Device Selectors. Since the HP-IB allows the interconnection of several devices, each device must have a means of being uniquely accessed. Specifying just the interface select code of the HP-IB interface through which a device is connected to the computer is not sufficient to uniquely identify a specific device on the bus.

Each device on the bus has a primary address by which it is identified. This address must be unique to allow individual access of each device. Each HP-IB device has a set of switches that are used to set its address. Thus, when a particular HP-IB device is to be accessed, it must be identified with both its interface select code and its bus address.

The interface select code is the first part of an HP-IB device selector. The interface select code of the internal HP-IB is 7; external interfaces can range from 8 to 31. The second part of an HP-IB device selector is the device's primary address, which are in the range of 0 through 30. For example, to specify the device:

On interface select code 7 with primary address 22   Use device selector = 722

On interface select code 10 with primary address 2   Use device selector = 1002

Remember that each device's address must be unique. The procedure for setting the address of an HP-IB device is given in the installation manual for each device. The HP-IB interface also has an address. The default address of the internal HP-IB is 21 or 20, depending on whether or not it is a System Controller, respectively. The addresses of an external HP-IB interface's address can be determined by reading STATUS register 3 of the appropriate interface select code, and each interface's address can be changed by writing to CONTROL register 3. See "Determining Controller Status and Address" and "Changing the Controller's Address" for further details.

Moving Data Through the HP-IB. Data is output from and entered into the computer through the HP-IB with the OUTPUT and ENTER statements, respectively. The only difference between the OUTPUT and ENTER statements for the HP-IB and those for other interfaces is the addressing information within HP-IB device selectors.

Examples

```
100   Hpib=7
110   Device_addr=22
120   Device_selector=Hpib*100+Device_addr
130   !
140   OUTPUT Device_selector;"FIR7T2T3"
150   ENTER Device_selector;Reading

320   ASSIGN @Hpib_device TO 702
330   OUTPUT @Hpib_device;"Data message"
340   ENTER @Hpib_device;Number

440   OUTPUT 822;"FIR7T2T3"

380   ENTER 724;Readings(*)
```

General Structure of the HP-IB. Communications through the HP-IB are made according to a precisely defined set of rules. These rules help to ensure that only orderly communication may take place on the bus. For conceptual purposes, the organization of the HP-IB can be compared to that of a committee. A committee has certain "rules of order" that govern the manner in which business is to be conducted. For the HP-IB, these rules of order are the IEEE 488-1978 standard.

On the HP-IB, the System Controller corresponds to the committee chairman. The system controller is generally designated by setting a switch on the interface and cannot be changed under program control. However, it is possible to designate an "acting chairman" on the HP-IB. On the HP-IB, this device is called the Active Controller, and may be any device capable of directing HP-IB activities, such as a desktop computer.

When the System Controller is first turned on or reset, it assumes the role of Active Controller. Thus, only one device can be designated System Controller. These responsibilities may be subsequently passed to another device while the System Controller tends to other business. This ability to pass control allows more than one computer to be connected to the HP-IB at the same time.

In a committee, only one person at a time may speak. It is the chairman's responsibility to "recognize" which one member is to speak. Usually, all committee members present always listen; however, this is not always the case on the HP-IB. One of the most powerful features of the bus is the ability to selectively send data to individual (or groups of) devices. This allows fast talkers to communicate with fast listeners without having to wait.

During a committee meeting, the current chairman is responsible for telling the committee which member is to be the talker and which is (are) to be the listener(s). Before these assignments are given, he must get the attention of all members. The talker and listener(s) are then designated, and the next data message is presented to the listener(s) by the talker. When the talker has finished the message, the designation process may be repeated.

On the HP-IB, the Active Controller takes similar action. When talker and listener(s) are to be designated, the attention signal line (ATN) is asserted while the talker and listener(s) are being addressed. ATN is then cleared, signaling that those devices not addressed to listen may ignore all subsequent data messages. Thus, the ATN line separates data from commands; commands are accompanied by the ATN line being true, while data messages are sent with the ATN line false.

On the HP-IB, devices are addressed to talk and addressed to listen in the following orderly manner. The Active Controller first sends a single command which causes all devices to unlisten. The talker's address is then sent, followed by the address(es) of the listener(s). After all listeners have been addressed, the data can be sent from the talker to the listener(s). Only device(s) addressed to listen accept any data that is sent through the bus (until the bus is reconfigured by subsequent addressing commands).

The data transfer, or data message, allows for the exchange of information between devices on the HP-IB. Our committee conducts business by exchanging ideas and information between the speaker and those listening to his presentation. On the HP-IB, data is transferred from the active talker to the active listener(s) at a rate determined by the slowest active listener on the bus. This restriction on the transfer rate is necessary to ensure that no data is lost by any device addressed to listen. The handshake used to transfer each data byte ensures that all data output by the talker is received by all active listeners.

Examples of Bus Sequences. Most data transfers through the HP-IB involve a talker and only one listener. For instance, when an OUTPUT statement is used (by the Active Controller) to send data to an HP-IB device, the following sequence of commands and data is sent through the bus.

```
OUTPUT 701;"Data"
```

1. The talker's address is sent (here, the address of the computer; "My Talk Address"), which is also a command.

2. The unlisten command is sent.

3. The listener's address (01) is sent, which is also a command.

4. The data bytes "D", "a", "t", "a", CR, and LF are sent; all bytes are sent using the HP-IB's interlocking handshake to ensure that the listener has received each byte.

Similarly, most ENTER statements involve transferring data from a talker to only one listener. For instance, the following ENTER statement invokes the following sequence of commands and data-transfer operations.

```
ENTER 722;Voltage
```

1. The talker's address (22) is sent, which is a command.

2. The unlisten command is sent.

3. The listener's address is sent (here, the computer's address; "My Listen Address"), also a command

4. The data is sent by device 22 to the computer using the HP-IB handshake.

Addressing Multiple Listeners. HP-IB allows more than one device to listen simultaneously to data sent through the bus (even though the data may be accepted at differing rates). The following examples show how the Active Controller can address multiple listeners on the bus.

```
100   ASSIGN @Listeners TO 701,702,703
110   OUTPUT @Listeners;String$
120   OUTPUT @Listeners USING Image_1;Array$(*)
```

This capability allows a single OUTPUT statement to send data to several devices simultaneously. It is however, necessary for all the devices to be on the same interface. When the preceding OUTPUT statement is executed, the unlisten command is sent first, followed by the Active Controller's talk address and then listen addresses 01, 02, and 03. Data is then sent by the controller and accepted by devices at addresses 1, 2, and 3.

If an ENTER statement that uses the same I/O path name is executed by the Active Controller, the first device is addressed as the talker (the source of data) and all the rest of the devices, including the Active Controller, are addressed as listeners. The data is then sent from the device at address 01 to the devices at addresses 02 and 03 and to the Active Controller.

```
130   ENTER @Listeners;String$
140   ENTER @Listeners USING Image_2;Array$(*)
```

Secondary Addressing. Many devices have operating modes which are accessed through the extended addressing capabilities defined in the bus standard. Extended addressing provides for a second address parameter in addition to the primary address. Examples of statements that use extended addressing are as follows.

```
100   ASSIGN @Device TO 72205   ! 22=primary, 05=secondary.
110   OUTPUT @Device;Message$

200   OUTPUT 72205;Message$

150   ASSIGN @Device TO 7220529 ! Additional secondary
160
170   OUTPUT @Device;Message$

120   OUTPUT 7220529;Message$
```

The range of secondary addresses is 00-31; up to six secondary addresses may be specified (a total of 15 digits including interface select code and primary address). Refer to the device's operating manual for programming information associated with the extended addressing capability. The HP-IB interface also has a mechanism for detecting secondary commands.

General Bus Management

The HP-IB standard provides several mechanisms that allow managing the bus and the devices on the bus. Here is a summary of the statements that invoke these control mechanisms.

ABORT is used to abruptly terminate all bus activity and reset all devices to power-on states.

CLEAR is used to set all (or only selected) devices to a predefined, device-dependent state.

LOCAL is used to return all (or selected) devices to local (front-panel) control.

LOCAL LOCKOUT is used to disable all devices' front-panel controls.

PPOLL is used to perform a parallel poll on all devices (which are configured and capable of responding).

PPOLL CONFIGURE is used to setup the parallel poll response of a particular device.

PPOLL UNCONFIGURE is used to disable the parallel poll response of a device (or all devices on an interface).

REMOTE is used to put all (or selected) devices into their device-dependent, remote modes.

SEND is used to manage the bus by sending explicit command or data messages.

SPOLL is used to perform a serial poll of the specified device (which must be capable of responding).

TRIGGER is used to send the trigger message to a device (or selected group of devices).

These statements (and functions) are described in the following discussion. However, the actions that a device takes upon receiving each of the above commands are, in general, different for each device. Refer to a particular device's manuals to determine how it will respond.

Remote Control of Devices. Most HP-IB devices can be controlled either from the front panel or from the bus. If the device's front panel controls are currently functional, it is in the Local state. If it is being controlled through the HP-IB, it is in the Remote state. Pressing the front-panel "Local" key will return the device to Local (front-panel) control, unless the device is in the Local Lockout state (described in a subsequent discussion).

The Remote message is automatically sent to all devices whenever the System Controller is powered on, reset, or sends the Abort message. A device also enters the Remote state automatically whenever it is addressed. The REMOTE statement also outputs the Remote message, which causes all (or specified) devices on the bus to change from local control to remote control. The computer must be the System Controller to execute the REMOTE statement.

Examples

```
REMOTE 7

ASSIGN @Device TO 700
REMOTE @Device

REMOTE 700
```

Locking Out Local Control. The Local Lockout message effectively locks out the "local" switch present on most HP-IB device front panels, preventing a device's user from interfering with system operations by pressing buttons and thereby maintaining system integrity. As long as Local Lockout is in effect, no bus device can be returned to local control from its front panel.

The Local Lockout message is sent by executing the LOCAL LOCKOUT statement. This message is sent to all devices on the specified HP-IB interface, and it can only be sent by the computer when it is the Active Controller.

Examples

```
ASSIGN @Hpib TO 7
LOCAL LOCKOUT @Hpib

LOCAL LOCKOUT 7
```

The Local Lockout message is cleared when the Local message is sent by executing the LOCAL statement. However, executing the ABORT statement does not cancel the Local Lockout message.

Enabling Local Control. During system operation, it may be necessary for an operator to interact with one or more devices. For instance, an operator might need to work from the front panel to make special tests or to troubleshoot. And, in general, it is good systems practice to return all devices to local control upon conclusion of remote-control operation. Executing the LOCAL statement returns the specified devices to local (front-panel) control. The computer must be the Active Controller to send the LOCAL message.

Examples

```
ASSIGN @Hpib TO 7
LOCAL @Hpib

ASSIGN @Device TO 700
LOCAL @Device
```

If primary addressing is specified, the Go-to-Local message is sent only to the specified device(s). However, if only the interface select code is specified, the Local message is sent to all devices on the specified HP-IB interface and any previous Local Lockout message (which is still in effect) is automatically cleared. The computer must be the System Controller to send the Local message (by specifying only the interface select code).

Triggering HP-IB Devices. The TRIGGER statement sends a Trigger message to a selected device or group of devices. The purpose of the Trigger message is to initiate some device-dependent action; for example, it can be used to trigger a digital voltmeter to perform its measurement cycle. Because the response of a device to a Trigger Message is strictly device-dependent, neither the Trigger message nor the interface indicates what action is initiated by the device.

Examples

```
ASSIGN @Hpib TO 7
TRIGGER @Hpib

ASSIGN @Device TO 707
TRIGGER @Device
```

Specifying only the interface select code outputs a Trigger message to all devices currently addressed to listen on the bus. Including device addresses in the statement triggers only those devices addressed by the statement. The computer can also respond to a trigger from another controller on the bus.

Clearing HP-IB Devices. The CLEAR statement provides a means of "initializing" a device to its predefined, device-dependent state. When the CLEAR statement is executed, the Clear message is sent either to all devices or to the specified device(s), depending on the information contained within the device selector. If only the interface select code is specified, all devices on the specified HP-IB interface are cleared. If primary-address information is specified, the Clear message is sent only to the specified device. Only the Active Controller can send the Clear message.

Examples

```
ASSIGN @Hpib TO 7
CLEAR @Hpib
ASSIGN @Device TO 700
CLEAR @Device
```

Aborting Bus Activity. This statement may be used to terminate all activity on the bus and return all the HP-IB interfaces of all devices to a reset (or power-on) condition. Whether this affects other modes of the device depends on the device itself. The computer must be either the active or the system controller to perform this function. If the System Controller (which is not the current Active Controller) executes this statement, it regains active control of the bus. Only the interface select code may be specified; device selectors which contain primary-addressing information (such as 724) may not be used.

Examples

```
ASSIGN @Hpib TO 7
ABORT @Hpib

ABORT 7
```

HP-IB Service Requests. Most HP-IB devices, such as voltmeter, frequency counters, and spectrum analyzers, are capable of generating a "service request" when they require the Active Controller to take action. Service requests are generally made after the device has completed a task (such as making a measurement) or when an error condition exists (such as a printer being out of paper). The operation and/or programming manuals for each device describes the device's capability to request service and conditions under which the device will request service.

To request service, the device sends a Service Request message (SRQ) to the Active Controller. The mechanism by which the Active Controller detects these requests is the SRQ interrupt. Interrupts allow an efficient use of system resources, because the system may be executing a program until interrupted by an event's occurrence. If enabled, the external event initiates a program branch to a routine which "services" the event (executes remedial action).

Setting Up and Enabling SRQ Interrupts. In order for an HP-IB device to be able to initiate a service routine as the Active Controller, two prerequisites must be met: the SRQ interrupt event must have a service routine defined, and the SRQ interrupt must be enabled to initiate the branch to the service routine. The following program segment shows an example of setting up and enabling an SRQ interrupt.

```
100   Hpib=7
110   ON INTR Hpib GOSUB Service_routine
120   !
130   Mask=2   ! Bit 1 enables SRQ interrupts.
140   ENABLE INTR Hpib;Mask
```

The value of the mask in the ENABLE INTR statement determines which type(s) of interrupts are to be enabled. The value of the mask is automatically written into the HP-IB interface's interrupt-enable register (CONTROL register 4) when this statement is executed. Bit 1 is set in the preceding example, enabling SRQ interrupts to initiate a program branch. Reading STATUS register 4 at this point would return a value of 2.

Servicing SRQ Interrupts. The SRQ is a level-sensitive interrupt; in other words, if an SRQ is present momentarily but does not remain long enough to be sensed by the computer, the interrupt will not be generated.

It is important to note that once an interrupt is sensed and logged, the interface cannot generate another interrupt until the initial interrupt is serviced. The computer disables all subsequent interrupts from an interface until a pending interrupt is serviced. For this reason, it was necessary to re-enable the interrupt to allow for subsequent branching.

Polling HP-IB Devices. The Parallel Poll is the fastest means of gathering device status when several devices are connected to the bus. Each device (with this capability) can be programmed to respond with one bit of status when Parallel Polled, making it possible to obtain the status of several devices in one operation. If a device responds affirmatively ("I need service") to a Parallel Poll, more information as to its specific status can be obtained by conducting a Serial Poll of the device.

Configuring Parallel Poll Responses. Certain devices can be remotely programmed by the Active Controller to respond to a Parallel Poll. A device which is currently configured for a Parallel Poll responds to the poll by placing its current status on one of the bus data lines. The logic sense of the response and the data-bit number can be programmed by the PPOLL CONFIGURE statement. No multiple listeners can be specified in the statement; if more than one device is to respond on a single bit, each device must be configured with a separate PPOLL CONFIGURE statement.

Conducting a Parallel Poll. The PPOLL function returns a single byte containing up to 8 status bit messages of all devices on the bus capable of responding to the poll. Each bit returned by the function corresponds to the status bit of the device(s) configured to respond to the parallel poll. (Recall that one or more devices can respond on a single line.) The PPOLL function can only be executed when the computer is the Active Controller.

Example

```
Response=PPOLL(7)
```

Disabling Parallel Poll Responses. The PPOLL UNCONFIGURE statement gives the computer (as Active Controller) the capability of disabling the Parallel Poll responses of one or more devices on the bus.

Examples. The following statement disables device 5 only.

```
PPOLL UNCONFIGURE 705
```

This statement disables all devices on interface select code 8 from responding to a Parallel Poll.

```
PPOLL UNCONFIGURE 8
```

If no primary addressing is specified, all bus devices are disabled from responding to a Parallel Poll. If primary addressing is specified, only the specified devices (which have the Parallel Poll Configure capability) are disabled.

Conducting a Serial Poll. A sequential poll of individual devices on the bus is known as a Serial Poll. One entire byte of status is returned by the specified device in response to a Serial Poll. This byte is called the Status Byte message and, depending on the device, may indicate an overload, a request for service, or a printer being out of paper. The particular response of each device depends on the device.

The SPOLL function performs a Serial Poll of the specified device; the computer must be the Active Controller.

Examples

```
ASSIGN @Device TO 700
Status_byte=SPOLL(@Device)

Spoll_24=SPOLL(724)
```

Just as the Parallel Poll is not defined for individual devices, the Serial Poll is meaningless for an interface; therefore, primary addressing must be used with the SPOLL function.

The Computer As a Non-Active Controller

The section called "General Structure of the HP-IB" described how communications take place through HP-IB Interfaces. The functions of the System Controller and Active Controller were likened to a "committee chairman" and "acting chairman," respectively, and the functions of each were described. This section describes how the Active Controller may "pass control" to another controller and assume the role of a non-Active Controller. This action is analogous to designating another committee member to take the responsibility of acting chairman and then becoming a committee member who listens to the acting chairman and speaks when given the floor.

Determining Controller Status and Address. It is often necessary to determine if an interface is the System Controller and to determine whether or not it is the current Active Controller. It is also often necessary to determine or change the interface's primary address.

Example. Executing the following statement reads STATUS register 3 (of the internal HP-IB) and places the current value into the variable Stat_and_addr. Remember that if the statement is executed from the keyboard, the variable Stat_and_addr must be defined in the current context.

```
STATUS 7,3;Stat_and_addr
```

Status Register 3:
Controller Status
and Address

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
System Controller	Active Controller	0	Primary Address of Interface				
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

If bit 7 is set (1), it signifies that the interface is the System Controller; if clear (0), it is not the System Controller. Only one controller on each HP-IB interface should be configured as the System Controller.

If bit 6 is set (1), it signifies that the interface is currently the Active Controller; if it is clear (0), another controller is currently the Active Controller.

Bits 4 through 0 represent the current value of the interface's primary address, which is in the range of 0 through 30. The power-on default value for the internal HP-IB is 21 (if it is the System Controller) and 20 (if not the System Controller). For external HP-IB interfaces, the default address is set by the configuration file. (Refer to appendix F).

Example. Calculate the primary address of the interface from the value previously read from STATUS register 3.

```
Intf_addr=Stat_and_addr MOD 32
```

This numerical value corresponds to the talk (or listen) address sent by the computer when an OUPUT (or ENTER) statement containing primary-address information is executed.

Changing the Controller's Address. It is possible to use the CONTROL statement to change an HP-IB interface's address.

Example

```
CONTROL 7,3;Intf_addr
```

The value of Intf_addr is used to set the address of the HP-IB interface (in this case, the internal HP-IB). The valid range of addresses is 0 through 30; address 31 is not used. Thus, if a value greater than 30 is specified, the value MOD 32 is used (for example: 32 MOD 32 equals 0, 33 MOD 32 equals 1, 62 MOD 32 equals 30, and so forth).

Passing Control. The current Active Controller can pass this capability to another computer by sending the Take Control message (TCT). The Active Controller must first address the prospective new Active Controller to talk, after which the TCT message is sent. If the other controller accepts the message, it then assumes the role of Active Controller; this computer then assumes the role of a non-Active Controller.

Passing control can be accomplished in one of two ways: it can be handled by the system, or it can be handled by the program. The PASS CONTROL statement can be used. For example, the following statements first define the HP-IB Interface's select code and new Active Controller's primary address and then pass control to that controller.

```
100 Hp_ib=7
110 New_ac_addr=20
120 PASS CONTROL 100*Hp_ib+New_ac_addr
```

The following statements perform the same functions.

```
100 Hp_ib=7
110 New_ac_addr=20
120 SEND Hp_ib;TALK New_ac_addr CMD 9
```

Once the new Active Controller has accepted the TCT command, the controller passing control assumes the role of a non-Active Controller (or "HP-IB device") on the specified HP-IB Interface. The next section describes the responsibilities of the computer while it is a non-Active Controller.

Interrupts While Non-Active Controller. When the computer is not an Active Controller, it must be able to detect and respond to many types of bus messages and events.

The computer (as a non-Active Controller) needs to keep track of the following information.

- It must keep track of itself being addressed as a listener so that it can enter data from the current active talker.
- It must keep track of itself being addressed as a talker so that it can transmit the information desired by the active controller.
- It must keep track of being sent a Clear, Trigger, Local, or Local Lockout message so that it can take appropriate action.
- It must keep track of control being passed from another controller.

One way to do this is to continually monitor the HP-IB interface by executing the STATUS statement and then taking action when the values returned match the values desired. This is obviously a great waste of computer time if the computer could be performing other tasks. Instead, the interface hardware can be enabled to monitor bus activity and then generate interrupts when certain events take place.

The computer has the ability to keep track of the occurrences of all of the preceding events. In fact, it can monitor up to 16 different interrupt conditions. STATUS registers 4, 5 and 6 provide access to the interface state and interrupt information necessary to design very powerful systems with a great degree of flexibility.

Each individual bit of STATUS register 4 corresponds to the same bit of STATUS register 5. Register 4 provides information as to which condition caused an interrupt, while register 5 keeps track of which interrupt conditions are currently enabled. To enable a combination of conditions, add the decimal values for each bit that you want set in the interrupt-enable register. This total is then used as the mask parameter in an ENABLE INTR statement.

Status Register 5: Interrupt Enable Mask

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	Hand-shake Error	Unrecognized Universal Command	Secondary Command While Addressed	Clear Received	Unrecognized Addressed Command	SRQ Received	IFC Received
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Bit 15 enables an interrupt upon becoming the Active Controller. The computer then has the ability to manage bus activities.

Bit 14* enables an interrupt upon detecting a change in Parallel Poll Configuration.

Bit 13 enables an interrupt upon being addressed as an active talker by the Active Controller.

Bit 12 enables an interrupt upon being addressed as an active listener by the Active Controller.

Bit 11 enables an interrupt when an EOI is received during an ENTER operation (the EOI signal line is also described in "HP-IB Control Lines").

Bit 10 enables an interrupt when the Active Controller performs a Serial Poll on the computer (in response to its service request).

Bit 9 enables an interrupt upon receiving either the Remote or the Local message from the active controller, if addressed to listen. The action taken by the computer is, of course, dependent on the user-programmed service routine.

Bit 8 enables an interrupt upon a change in talk or listen address. An interrupt will be generated if the computer is addressed to listen or talk or "idled" by an Unlisten or Untalk command.

Bit 7 enables an interrupt upon receiving a Trigger message, if the computer is currently addressed to listen. This interrupt can be used in situations where the computer may be "armed and waiting" to initiate action; the active controller sends the Trigger message to the computer to cause it to begin its task.

---

* This condition requires accepting data from the bus and then explicitly releasing the bus.

Bit 6 enables an interrupt if a bus error occurs during an OUTPUT statement. Particularly, the error occurs if none of the devices on the bus respond to the HP-IB's interlocking handshake (see "HP-IB Control Lines"). The error typically indicates that either a device is not connected or that its power is off.

Bit 5* enables an interrupt upon receiving an unrecognized Universal Command. This interrupt condition provides the computer with the capability of responding to new definitions that may be adopted by the IEEE standards committee.

Bit 4* enables an interrupt upon receiving a Secondary Command (extended addressing) after the interface receives either its primary talk address or primary listen address. Again, this interrupt provides the computer with a way to detect and respond to special messages from another controller.

Bit 3 enables an interrupt on receiving a Clear message. Reception of either a Device Clear message (addressed to the computer) will cause this type of interrupt. The computer is free to take any "device-dependent" action, such as setting up all default values again or even restarting the program, if that is defined by the programmer to be the "cleared" state of the machine.

Bit 2* enables an interrupt upon receiving an unrecognized Addressed Command, if the computer is currently addressed to listen. This interrupt is used to intercept and respond to bus commands which are not defined by the standard.

Bit 1 enables an interrupt upon detecting a Service Request.

Bit 0 enables an interrupt upon detecting an Interface Clear (IFC). The interrupt is generated only when the computer is not the System Controller, as only a System Controller is allowed to set the Interface Clear signal line. The service routine typically is used to recover from the abrupt termination of an I/O operation caused by another controller sending the IFC message.

* This condition requires accepting data from the bus and then explicitly releasing the bus. Refer to the "Advanced Bus Management" section for further details.

Note that most of the conditions are state-sensitive or event-sensitive; the exception is the SRQ event, which is level-sensitive. State or event-sensitive events can never go unnoticed by the computer as can service requests; the event's occurrence is "remembered" by the computer until serviced.

For instance, if the computer is enabled to generate an interrupt on becoming addressed as a talker, it would interrupt the first time it received its own talk address. After having responded to the service request (most likely with some sort of OUTPUT operation), it would not generate another interrupt, even if it was still left assigned as a talker by the Active Controller. Thus, it would not generate another interrupt until the event occurred a second time.

An oversimplified example of a service routine that is to respond to multiple conditions might be as follows. This example can be found in file SERVER1 on your Manual Examples disc.

Register 4, the interrupt status register, is a "read-destructive" register; reading the register with a STATUS statement returns its contents and then clears the register (to a value of 0). If the service routine's action depends on the contents of STATUS register 4, the variable in which it is stored must not be used for any other purposes before all of the information that it contains has been used by the service routine.

The computer is automatically addressed to talk (by the Active Controller) whenever it is Serially Polled. If interrupts are concurrently enabled for My Address Change and/or Talker Active, the ON INTR branch will be initiated due to the reception of the computer's talk address. However, since the Serial Poll is automatically finished with the Untalk Command, the computer may no longer be addressed to talk by the time the interrupt service routine begins execution. See "Responding to Serial Polls" for further details.

Requesting Service. When the computer is a non-Active Controller, it has the capability of sending an SRQ to the current Active Controller. The following statement is an example of requesting service from the Active Controller of the HP-IB Interface on select code 7.

```
CONTROL 7,1;64
```

The REQUEST statement can be used to perform the same function.

```
REQUEST 7;64
```

Both of the preceding examples place a logic True on the SRQ line. (Note that the line may already be set True by another device.) Other bits may be set in the Status Byte message, indicating that other device-dependent conditions exist.

The SRQ line is held True until the Active Controller executes a Serial Poll or this computer executes a REQUEST with bit 6 equal to 0. (Note also that the line may still be held True by another device.)

Responding to Parallel Polls. Before performing a Parallel Poll of bus devices, the Active Controller configures selected device(s) to respond on one of the eight data lines. Each device is directed to respond on a particular data line with a logic True or False; the logic sense of the response informs the Active Controller either "I do need service" or "I don't need service." The logic sense of the response is also specified by the Active Controller. This response to the Parallel Poll is known as the Status Bit message.

Responding to Serial Polls. As a non-Active Controller, the response to Serial Polls is automatically handled by the system. The desired Serial Poll Response Byte is sent to HP-IB CONTROL Register 1. If bit 6 is set (bit 6 has a value of 64), an SRQ is indicated from this controller. All other bits can be considered to be "device-dependent," and can thus be set according to the program's needs.

The following statement sets up a response with SRQ and bits 1 and 0 set to 1.

```
CONTROL 7,1;64+2+1
```

When the Active Controller performs a Serial Poll on this non-Active Controller, the specified byte is automatically sent to the Active Controller by the system.

The RS-232 Serial Interface

The Serial Interface is an RS-232-C compatible interface used for simple asynchronous I/O applications such as driving line printers, terminals, or other peripherals. It uses a UART (Universal Asynchronous Receiver and Transmitter) integrated circuit to generate the required async signals. The computer must provide most control functions because the card does not have its own processor capability. Consequently, there is more interaction between the card and computer than when you use a more intelligent interface except for relatively simple applications.

The RS-232-C interface standard establishes electrical and mechanical interface requirements, but does not define the exact function of all the signals that are used by various manufacturers of data communications equipment and serial I/O devices. Consequently, when you plug your serial interface into an RS-232 connector, there is no guarantee the devices can communicate unless you have configured optional parameters to match the requirements of the device you are connecting to.

Note

RS-232-C is a data communication standard established and published by the Electronic Industries Association (EIA). Copies of the standard are available from the association at 2001 Eye Street N.W., Washington D.C. 20006. Its equivalent for European applications is CCITT V.24.

Asynchronous Data Communication

The terms Asynchronous (Async for short) data communication and Serial I/O refer to a technique of transferring information between two communicating devices by means of bit-serial data transmission. This means that data is sent, one bit at a time, and that characters are not synchronized with preceding or subsequent data characters; that is, each character is sent as a complete entity without relationship to other events, before or after. Characters may be sent in close succession, or they may be sent sporadically as data becomes available. Start and stop bits are used to identify the beginning and end of each character, with the character data placed between them.

Character Format. Each character frame consists of the following elements:

- Start Bit: The start bit signals the receiver that a new character is being sent. Since the receiver knows how many bits per second are being transmitted (specified by the baud rate), it can determine the expected arrival time for all subsequent bits in that character frame. All other bits in a given frame are synchronized to the start bit.

- 5–8 Character Data Bits: The next bits are the binary code of the character being transmitted, consisting of 5, 6, 7, or 8 bits; depending on the application. The parity bit is not included in the character data bits.

- Parity Bit: The parity bit is optional, included only when parity is enabled.

- Stop Bit(s): One or more stop bits identify the end of each character. The serial interface has no provision for inserting the time gaps between characters.

Here is a simple diagram showing the structure of an asynchronous character and its relationship to other characters in the data stream:
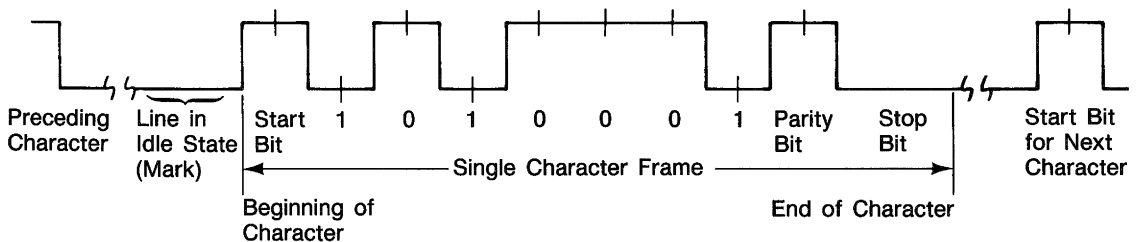
Parity. The parity bit is used to detect errors as incoming characters are received. If the parity bit does not match the expected sense, the character is assumed to be incorrectly received. The action taken when an error is detected depends upon how the interface and your computer program are configured.

Parity sense is determined by system requirements. The parity bit may be included or omitted from each character by enabling or disabling the parity function. If the parity bit is enabled, four options are available. Parity is checked by the receiver for all parity options including ONE and ZERO. Parity options include:

- NONE   Parity function is DISABLED, and the parity bit is omitted from each character frame.

- ODD   Parity bit is SET if there is an EVEN* number of ones in the data character. The receiver performs parity checks on incoming character.

- EVEN   Parity bit is SET if there is an ODD* number of ones in the data character. The receiver performs parity checks on incoming characters.

- ONE   Parity bit is set for all characters. Parity is checked by the receiver on all incoming characters.

- ZERO   Parity bit is cleared, but present for all characters. Parity is checked by the receiver on all characters.

---

* Parity sense is determined by counting the number of ones in the character INCLUDING the parity bit. Consequently, the parity sense is reversed from the number of ones in a character without the parity bit.

Error Detection. Two types of incoming data errors can be detected by serial receivers:

- Parity errors are signalled when the parity bit does not match the number of ones, including the parity bit, even or odd as defined by interface configuration. When parity is disabled, no parity check is made.

- Framing errors are signalled when start and stop bits are not properly received during the expected time frame. They can be caused by a missing start bit, noise errors near the end of the character, or by improperly specified character length at the transmitter or receiver.

Two additional error types are detected by the receiver section of the serial interface:

- Overrun errors result when the desktop computer does not consume characters as fast as they arrive. The card provides only one character of buffer space, so the current character must be consumed by an ENTER before the next character arrives. Otherwise, the character is lost when the next character replaces it, and an error is sent to BASIC.

- Received BREAKs are detected as a special type of framing error. They generate the same type of BASIC error as framing errors.

Data Transfers Between Computer and Peripheral

Four statements are used to transfer information between your computer and the interface card:

- The CONTROL statement is used to control interface operation and defines such parameters as baud rate, character format, or parity.

- The OUTPUT statement sends data to the interface which, in turn, sends the information to the peripheral device.

- The ENTER statement inputs data from the interface card after the interface has received it from the peripheral device.

- The STATUS statement is used to monitor the interface and obtain informaion about interface operation such as buffer status, detected errors, and interrupt enable status.

Since the interface has no on-board processor, ENTER and OUTPUT statements cause the computer to wait until the ENTER or OUTPUT operation is complete before continuing to the next line. For OUTPUT statements, this means that the computer waits until the last bit of the last character has been sent over the serial line before continuing with the next program statement.

Overview of Serial Interface Programming

Serial interface programming techniques are similar to most general I/O applications. The interface card is initialized by use of CONTROL statements; STATUS statements evaluate its readiness for use. Data is transferred between your computer and a peripheral device by OUTPUT and ENTER statements. In most cases, you can use default configuration switches on the interface card to eliminate or significantly reduce the need for using CONTROL statements to initialize the card.

Due to the asynchronous nature of serial I/O operations, you should take special care to ensure that data is not lost by sending to a device before the device is ready to receive. Modem line handshaking can be used to help solve this problem. Refer to the *BASIC 5.0 Language Reference M–Z* (Volume 2) for a description of interface registers.

Initializing the Interconnection

Determining Operating Parameters. Before you can successfully transfer information to a device, you must match the operating characteristics of the interface to the corresponding characteristics of the peripheral device. This includes matching signal lines and their functions as well as matching the character format for both devices.

Hardware Parameters. To determine hardware operating parameters, you need to know the answer for each of the following questions about the peripheral device:

- Which of the following signal and control lines are actively used during communication with the peripheral?

  ___Data Set Ready (DSR)
  ___Data Carrier Detect (DCD or CD)
  ___Clear to Send (CTS)
  ___Ring Indicator (RI)

- What baud rate (line speed) is expected by the peripheral?

Character Format Parameters. To define the character format, you must know the requirements of the peripheral device for the following parameters:

- Character Length: How many data bits are used for each character, excluding start, stop, and parity bits?

- Parity Enable: Is Parity enabled (included) or disabled (absent) for each character?

- Parity Sense: Is the parity bit, if enabled, ODD, EVEN, always ONE, or always ZERO?

- Stop Bits: How many stop bits are included with each character: 1,1.5, or 2?

Using Interface Defaults to Simplify Programming. The serial interface may be preconfigured with default parameters. Refer to appendix F for details.

Using Program Control to Override Defaults

You can override some of the interface default configuration options by use of CONTROL statements. This not only enables you to guarantee certain parameters, but also provides a means for changing selected parameters in the course of a running program.

Interface Reset. Whenever an interface is connected to a modem that may still be connected to a telecommunications link from a previous session, it is good programming practice to reset the interface to force the modem to disconnect, unless the status of the link and remote connection are known. When the interface is connected to a line printer or similar peripheral, resetting the interface is usually unnecessary unless an error condition requires it.

When the interface is reset by use of a CONTROL statement to Control Register 0 with a non-zero value, the interface is restored to its default configuration, except that the current character format is not altered, whether or not it is the same as the current default configuration. If you are not sure of the present settings, or if your application requires changing the configuration during program operation, you can use CONTROL statements to configure the interface. An example of when this may be necessary is when several peripherals share a single interface through a manually operated RS-232 switch such as those used to connect multiple terminals to a single computer port, or a single terminal to multiple computers.

Selecting the Baud Rate. In order to successfully transfer information between the interface card and a peripheral, the interface and peripheral must be set to the same baud rate. A CONTROL statement to register 3 can be used to set the interface baud rate. To verify the current baud rate setting, use a STATUS statement addressed to register 3. All rates are in baud (bits/second).

Setting Character Format and Parity. Control Register 4 overrides the default configuration that controls parity and character format. To determine the value sent to the register, add the appropriate values selected from the following table:

Parity Sense	Parity Enable	Stop Bits	Character Length
0 ODD parity	0 Disabled	0 1 stop bit	0 5 bits/char
16 EVEN parity	8 Enabled	4 1.5 stop bits if 5 bits/char or 2 stop bits if 6,7,or 8 bits/char	1 6 bits/char
32 Always ONE			2 7 bits/char
48 Always ZERO			3 8 bits/char

For example, to configure a character format of 8 bits per character, two stop bits, and EVEN parity, use the following CONTROL statement:

```
1200 CONTROL Sc,4;3+4+8+16
```

or

```
1200 CONTROL Sc,4;31
```

To configure a 5-bit character length with 1 stop bit and no parity bit, use the following:

```
1200 CONTROL Sc,4;0
```

Data Transfers

The serial interface card is designed for relatively simple serial I/O operations. It is not intended for sophisticated applications that use ON INTR statements extensively to service the interface. Limited ON INTR capabilities are provided by the serial interface for error trapping and other simple tasks.

Program Flow. When the interface is properly configured, either by use of default switches or CONTROL statements, you are ready to begin data transfers. OUTPUT statements are used to send information to the peripheral; ENTER statements to input information from the external device. Any valid OUTPUT or ENTER statement and variable(s) list may be used, but you must be sure that the data format is compatible with the peripheral device. For example, non-ASCII data sent to an ASCII line printer results in unpredictable behavior.

Various other I/O statements can be used in addition to OUTPUT and ENTER, depending on the situation. For example, the LIST statement can be used to list programs to an RS-232 line printer PROVIDED the interface is properly configured before the operation begins.

Data Output. To send data to a peripheral, use OUTPUT, OUTPUT USING, or any other similar or equivalent construct. Suppression of end-of-line delimiters and other formatting capabilities are identical to normal operation in general I/O applications. The OUTPUT statement hangs the computer until the last bit of the last character in the statement variable list is transmitted by the interface. When the output operation is complete the computer then continues to the next line in the program.

Data Entry. To input data from a peripheral, use ENTER, ENTER USING, or an equivalent statement. Inclusion or elimination of end-of-line delimiters and other information is determined by the formatting specified in the ENTER statment. The ENTER statement hangs the computer until the input variables list is satisfied. To minimize the risk of waiting for another variable that isn't coming, you may prefer to specify only one variable for each ENTER statement, and analyze the result before starting the next input operation.

Be sure that the peripheral is not transmitting data to the interface while no ENTER is in progress. Otherwise, data may be lost because the card provides buffering for only one character. Also, interrupts from other I/O devices, or operator inputs to the computer keyboard can cause delays in computer service to the interface that result in buffer overrun at higher baud rates.

Modem Line Handshaking. Modem line handshaking, when used, is performed automatically by the computer as part of the OUTPUT or ENTER operation. After a given OUTPUT or ENTER operation is complete, the program continues execution on the next line.

Control Register 5 can be used to force selected modem control lines to their active state(s). The Data Rate Select and Secondary Request-to-Send lines are set or cleared by bits 3 and 2 respectively. Request-to-send and Data Terminal Ready are held in their active states when bits 1 and 0 are true, respectively. If bits 1 and/or 0 are false, the corresponding modem line is toggled during OUTPUT or ENTER as explained previously.

Incoming Data Error Detection and Handling. The serial interface card can generate several errors that are caused when certain conditions are encountered while receiving data from the peripheral device. The UART detects a given error condition and sets the corresponding bit in Status Register 10. The card then generates a pending error to BASIC.

Trapping Serial Interface Errors. Pending BASIC errors can be trapped by using an ON ERROR statement in conjunction with an error trapping service routine to evaluate the error condition.

The GPIO Interface

This section should be used in conjunction with the *HP 82306A GPIO Interface Installation Instructions*. The best way to use these two documents is to read this section before attempting to configure and connect the interface according to the directions given in the installation manual. The reason for this order of use is that knowing how the interface works and how it is driven by BASIC programs will help you to decide how to connect it to your peripheral device.

The GPIO Interface is a very flexible parallel interface that allows you to communicate with a variety of devices. The interface sends and receives up to 16 bits of data with a choice of several handshake methods. External interrupt and user-definable signal lines are provided for additional flexibility. The interface is known as the General-Purpose Input/Output (GPIO) Interface for these reasons. This section describes how to use the interface's features from BASIC Programs.

Use of some statements or suggestions for interfacing requires the TRANS binary file. If you do not have TRANS loaded, use LOAD BIN "TRANS" and STORE SYSTEM to load and store this binary.

Interface Description

The main function of any interface is obviously to transfer data between the computer and a peripheral device. This section briefly describes the interface lines and how they function.

The GPIO Interface provides 32 lines for data input and output: 16 for input (DI0-DI15), and 16 for output (DO0-DO15).
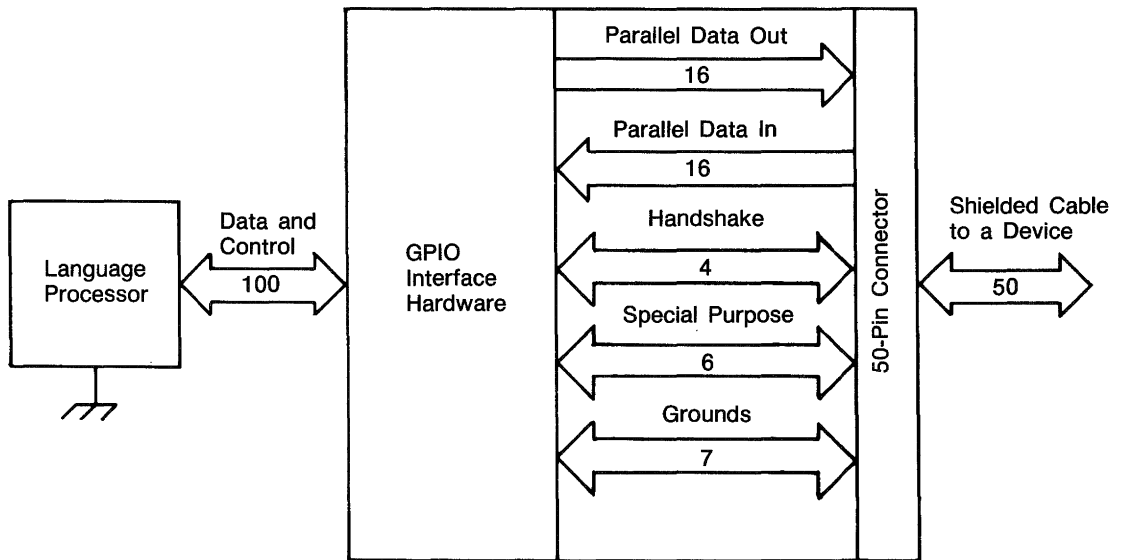
Three lines are dedicated to handshaking the data from source to destination device. The Peripheral Control line (PCTL) is controlled by the interface and is used to initiate data transfers. The Peripheral Flag line (PFLG) is controlled by the peripheral device and is used to signal the peripheral's readiness to continue the transfer process. The Input/Output line (I/O) is used to indicate direction of data flow.

One line is used to signal External Interrupt Requests to the computer (EIR). The interface must be enabled to initiate interrupt branches for the interface to detect this request. The state of the line can also be read by the program.

Four general-purpose lines are available for any purpose you desire; two are controlled by the computer and sensed by the peripheral (CTL0 and CTL1), and two are controlled by the peripheral device and sensed by the computer (STI0 and STI1).

Both Logic Ground and Safety Ground are provided by the interface. Logic Ground provides the reference point for signals, and Safety Ground provides earth ground for cable shields.

Interface Configuration

This section presents a brief summary of selecting the interface's configuration-switch settings. It is intended to be used as a checklist and to begin to acquaint you with programming the interface. Refer to the installation manual for the exact location and setting of each switch.

A sample program (found in file GPIOCHECK on your Manual Examples disc) checks a few of these switch settings on a GPIO Interface installed in the computer and displays the settings. However, many of the settings cannot be determined from BASIC programs. If any of the displayed settings are different than desired, or if any settings are not already known, refer to the installation manual for switch locations and settings.

Interface Select Code. In BASIC, allowable interface select codes range from 8 through 31; codes 1 through 7 are already used for built-in interfaces. The GPIO interface has a factory default setting of 12, which can be changed by re-configuring the "SEL CODE" switches on the interface.

Hardware Interrupt Priority. Two switches are provided on the interface to allow selection of hardware interrupt priority. The switches allow hardware priority level 3 through 6 to be selected. Hardware priority determines the order in which simultaneously occurring interrupt events are logged, while software priority determines the order in which interrupt events are serviced by the BASIC program.

Data Logic Sense. The data lines of the interface are normally low-true; in other words, when the voltage of a data line is low, the corresponding data bit is interpreted to be a 1. This logic sense may be changed to high-true with the Option Select Switch. Setting the switch labeled "DIN" to the "0" position selects high-true logic sense of Data In lines.
Conversely, setting the switch labeled "DOUT" to the "1" position inverts the logic sense of the Data Out lines. The default setting is "1" for both.

Data Handshake Methods. As a brief review, a data handshake is a method of synchronizing the transfer of data from the sending to the receiving device. In order to use any handshake method, the computer and peripheral device must be in agreement as to how and when several events will occur. With the GPIO Interface, the following events must take place to synchronize data transfers; the first two are optional.

- The computer may optionally be directed to perform a one-time "OK check" of the peripheral before beginning to transfer any data.

- The computer may also optionally check the peripheral to determine whether or not the peripheral is "ready" to transfer data.

- The computer must indicate the direction of transfer and then initiate the transfer.

- During OUTPUT operations, the peripheral must read the data sent from the computer while valid; similarly, the computer must clock the peripheral's data into the interface's Data In registers while valid during ENTER operations.

- The peripheral must acknowledge that it has received the data.

The GPIO handshakes data with three signal lines. The Input/Output line, I/O, is driven by the computer and is used to signal the direction of data transfer. The Peripheral Control line, PCTL, is also driven by the computer and is used to initiate all data transfers. The Peripheral Flag line, PFLG, is driven by the peripheral and is used to acknowledge the computer's requests to transfer data.

Handshake Logic Sense. Logic senses of the PCTL and PFLG lines are selected with switches of the same name. The logic sense of the I/O line is High for ENTER operations and Low for OUTPUT operations; this logic sense cannot be changed. The available choices of handshake logic sense and handshake modes allow nearly all types of peripheral handshakes to be accommodated by the GPIO Interface.

Handshake Modes. There are two general handshake modes in which the PCTL and PFLG lines may be used to synchronize data transfers: Full-Mode and Pulse-Mode Handshakes. If the peripheral uses pulses to handshake data transfers and meets certain hardware timing requirements, the Pulse-Mode Handshake may be used. The Full-Mode Handshake should be used if the peripheral does not meet the Pulse-Mode timing requirements.

The handshake mode is selected by the position of the "HSHK" switch on the interface, as described in the installation manual. Both modes are more fully described in subsequent sections.

Data-In Clock Source. Ensuring that the data are valid when read by the receiving device is slightly different for OUTPUT and ENTER operations. During OUTPUTs, the interface generally holds data valid while PCTL is in the Set state, so the peripheral must read the data during this period. During ENTERs, the data must be held valid by the peripheral until the peripheral signals that the data are valid (which clocks the data into interface Data In registers) or until the data is read by the computer. The point at which the data are valid is signalled by a transition of PFLG. The PFLG transition that is used to signal valid data is selected by the "CLK" switches on the interface. Subsequent diagrams and text further explain the choices.

Optional Peripheral Status Check. Many peripheral devices are equipped with a line which is used to indicate the device's current "OK-or-Not-OK" status. If this line is connected to the Peripheral Status line (PSTS) of the GPIO Interface, the computer may determine the status of the peripheral device by checking the state of the PSTS. The logic sense of this line may be selected by setting the "PSTS" switch. If the switch is enabled, the computer performs a one-time check of the Peripheral Status line (PSTS) before initiating any transfers as part of the data-transfer handshake. If PSTS indicates "Not OK," Error 172 is reported; otherwise, the transfer proceeds normally. If this feature is not enabled, this one-time check is never made. This feature is available with both Full-Mode and Pulse-Mode Handshakes.

Interface Reset

The interface should always be reset before use to ensure that it is in a known state. All interfaces are automatically reset by the computer at certain times: when the computer is powered on, when RESET is pressed. The interface may be optionally reset at other times under control of BASIC programs. Two examples are as follows:

```
Gpio=12
CONTROL Gpio,0;1

Reset=1
CONTROL Gpio;Reset
```

The following action is invoked whenever the GPIO Interface is reset:

- The Peripheral Reset line (PRESET) is pulsed Low for at least 15 microseconds.
- The PCTL line is placed in the Clear state.
- If the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logic 0).
- The interrupt enable bit is cleared, disabling subsequent interrupts until re-enabled by the program.

The following lines are unchanged by a reset of the GPIO Interface:

- The CTL0 and CTL1 output lines.
- The I/O line.
- The Data Out lines, if the DOUT CLEAR jumper is not installed.

Using OUTPUT and ENTER Through the GPIO

This section shows you how to use OUTPUT and ENTER through the GPIO Interface. The actual signals that appear on the data lines depend on three things: the data currently being transferred, how this data is being represented, and the logic sense of the data lines.

This section gives simple examples of how several representations are implemented during OUTPUTs and ENTERs through the GPIO Interface.

ASCII and Internal Representations. Data normally passes through the GPIO Interface one byte at a time, with the most significant byte first. This byte-mode transfer is independent of whether FORMAT ON or FORMAT OFF is the I/O path attribute.

Example Statements Using OUTPUT. The following examples show how you can use the OUTPUT statement to output data bytes through the GPIO interface.

```
ASSIGN @Gpio TO 12
OUTPUT @Gpio;"ASCII"

Gpio=12
Number=-4
OUTPUT Gpio USING "MD.DD";Number

ASSIGN @Gpio TO 12;FORMAT OFF
String$="1234"
OUTPUT @Gpio;String$
```

Example Statements Using ENTER. The following examples show how you can use the ENTER statement to enter data bytes through the GPIO interface.

```
ENTER @Gpio USING "#,B";Byte
DISP "Value Entered = ";Byte

Value Entered = 65

ENTER 12;String$
DISP "String Entered =

String Entered = ruok?

REAL Number
ASSIGN @Gpio TO 12
ENTER @Gpio;Number
DISP "Number = ;"Number

Number = 2
```

Example Statements that Output Data Words*.

```
Word=3*256+3
OUTPUT @Gpio USING "#,W";Output_word

Output_16_bits=-1
CONTROL Gp_isc,3;Output_16_bits
```

It is important to note that no output handshake is executed when the CONTROL statement is executed; only the states of the Data Out lines and the I/O lines are affected. Handshake sequence, if desired, must be performed by BASIC statements in the program.

---

* Data are automatically sent as words when using an I/O path with the WORD attribute.

Example Statements that Enter Data Words*.

```
ENTER 12 USING "#,W";Enter_16_bits
DISP "INTEGER entered = ";Enter_16_bits

INTEGER entered = 511

STATUS Gp_isc,3;Enter_16_bits
DISP "INTEGER entered = ";Enter_16_bits

INTEGER entered = -512
```

It is important to note that no enter handshake is performed when the STATUS statement is executed. The only actions taken are the I/O lines being placed in the High state and the Data In registers being read. If an enter handshake is required, it must be performed by the BASIC program.

Remember also that the Data In Clock source is solely determined by the switch setting on the interface card. Thus, when the STATUS statement is used to read the Data In lines, the data on the lines may or may not be clocked into the registers when the statement is executed. If the data are to be clocked in by the STATUS statement, the "READ" clock source must be selected. See the installation manual for further details.

GPIO Timeouts. This section explains how the time parameter is measured and describes typical service routines.

* Data are automatically received as words when using an I/O path with the WORD attribute.

Timeout Time Parameter. There are two general time intervals measured and compared to the specified TIMEOUT time. The first interval is measured between the computer initiating the first handshake (PCTL=Set) and the peripheral signalling Ready (with the PFLG line). If the peripheral does not indicate readiness by the specified TIMEOUT time parameter, a TIMEOUT event occurs.

The time elapsed during each handshake is also measured and compared to the TIMEOUT time. The timing begins when the transfer is initiated (PCTL Set by the computer) and, in general, ends when the peripheral responds on the PFLG line.

Keep in mind that the TIMEOUT time parameter specifies the minimum time that the computer will wait before initiating the ON TIMEOUT branch. However, the computer may occasionally wait an additional 25% of the specified time parameter before initiating the branch. For instance, if a time of 0.4 seconds is specified, the computer will wait at least 0.4 seconds for the handshake to be completed, but it may occasionally wait up to 0.5 seconds before initiating the ON TIMEOUT branch.

Timeout Service Routines. The service routine usually responds by determining if the peripheral is functioning properly ("OK") or is down ("not OK"). The simplest action that might be taken by the computer is to read the state of the PSTS signal line, as shown in the following service routine (found in file GPIOSERV on you Manuals Examples disc).

A TIMEOUT has been set up to occur if the peripheral takes approximately more than .08 seconds to complete its response during a data transfer; how the peripheral completes its response depends on the handshake mode currently selected. With Pulse-Mode Handshakes, the peripheral completes its response by using PFLG to Clear PCTL; with Full-Mode Handshakes, the response is complete only after PCTL has been Cleared and PFLG is in the Ready state.

When a TIMEOUT occurs, the computer automatically executes an Interface Reset; the PCTL line is Set and then Cleared, and the PRESET line is pulsed Low. See the section called "Interface Reset" for further effects. The Service routine checks the PSTS line to see if the peripheral is OK or not OK. If not OK, a message is displayed and the program is paused; if OK, program execution is returned to the line following that in which the TIMEOUT occurred. A service routine may be programmed to attempt the transfer again, if desired; however, the automatic Reset performed when the TIMEOUT occurred may make this type of response difficult to implement.

GPIO Interrupts

This section describes the types of and techniques for using the interrupts available on the GPIO Interface.

Types of Interrupt Events. The GPIO Interface can sense two interrupt events:

- The interface becoming "Ready" for subsequent handshakes.
- The External Interrupt Request Line (EIR) being driven to logic low by the peripheral.

Since both of these events initiate identical computer responses, the service routine must be able to determine which of these interrupts has occurred.

Setting Up and Enabling Events. When either event occurs, the interrupt is logged by the operating system. After logging the occurrence, any further interrupts from the GPIO Interface are automatically disabled until specifically enabled by a program. All further computer responses to either event depend entirely on the BASIC program currently in memory.

The following program segment shows the steps involved in setting up and enabling Ready Interrupts.

```
100  Gpio=12
110  ON INTR Gpio GOSUB Gpio_serv
120  !
130  Mask=2
140  ENABLE INTR Gpio;Mask
```

The value of the interrupt mask determines which, if any, of the GPIO interrupt events are to be enabled to initiate the corresponding branch. Bits of the Interrupt Mask register have the following definitions.

Interrupt Enable Register: (ENABLE INTR)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			Not Used			Enable Interface Ready Interrupts	Enable EIR Interrupts
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

Interface Ready. Setting this bit (1) enables an interrupt to initiate the ON INTR branch when the interface detects that it is Ready to handshake data. If Full-Mode Handshake is selected (with the Option Select switch), the Ready event is PCTL=Clear and PFLG=Ready. With Pulse-Mode Handshake, the event is PCTL=Clear (independent of the state of PFLG).

External Interrupt Request. Setting this bit (1) enables an interrupt to initiate the ON INTR branch when the interface senses an External Interrupt Request (EIR line=Low).

Interrupt Service Routines

If both events are enabled, the service routine must be able to differentiate between the two. And, if both have occurred, the service routine must be able to service both causes. The following registers contain the current state of the Interface Ready flag and EIR signal lines, from which the interrupt cause(s) may be determined.

Status Register 4: Interface Ready

The interface is ready for a subsequent data transfer; 1 = Ready, 0 = Busy.

Status Register 5: Peripheral Status

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	PSTS Ok	EIR Line Low	STI1 Line Low	STI0 Line Low
Value = 128	Value = 64	Value = 32	Value = 16	Value = 8	Value = 4	Value = 2	Value = 1

As mentioned in preceding paragraphs, these two interrupt causes are both level-sensitive events, not edge-triggered events. This fact has two important implications. The first is that, for an event to be recognized, the corresponding signal line must be held in the interrupting state until the computer can interrogate the line's logic state. If the signal line's state is changed before the service routine checks the line, the interrupt may be "missed". This will happen only if both events are enabled; if only one event is enabled, determining the cause may not be necessary.

The second implication is that the service routine must be able to acknowledge the request in order for the peripheral device to remove the request. If the request is not removed after service, the same request may be serviced more than once.

The program found in file EIRSERVE on your Manuals Examples disc shows a simple example of servicing an External Interrupt Request. Note that only EIR-type interrupts have been enabled and that the peripheral device provides its own interrupt cause with signals on the STI0 and STI1 lines.

A slightly different method that peripherals use to communicate the cause of their interrupt request is to place the interrupt cause on the data lines concurrent with the interrupt request. The service routine can determine the cause by reading STATUS register 3 and take the appropriate action.

Notice that the service routine indicates a likely place for a Ready-interrupt service routine. The Service routine must check for the Ready condition, acknowledge the interrupt, and then take the desired action. In this case, no service action has been defined because Ready interrupts have not been enabled. The next section provides an example of a Ready interrupt service routine.

The HP-HIL Interface

HP-HIL (Hewlett-Packard Human Interface Link) is an interface capable of supporting up to seven devices (such as a mouse, keyboard, or digitizer) generally related to human input. The following diagram illustrates the basic components of the HP-HIL interface.
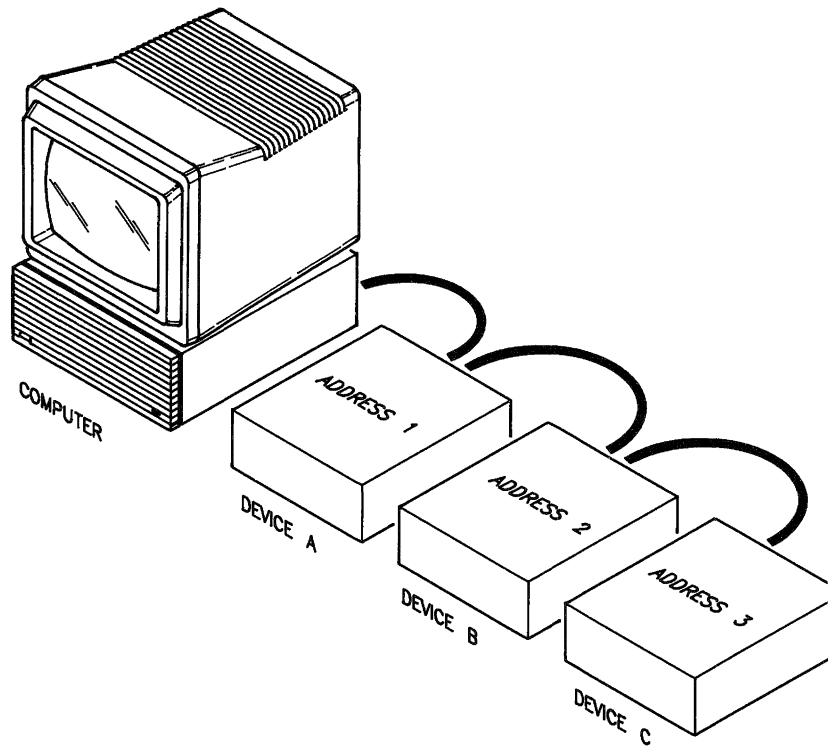
HP-HIL initialization occurs when you boot HP BASIC.
HP BASIC logs the HP-HIL devices present on the link. The
link can deal with a maximum of seven devices at a time. Any
devices added after the seventh are ignored. If you add a de-
vice to the link *after* HP Basic is booted, the device will not be
recognized by the system unless you boot HP BASIC again.
Also, if you replace an HP-HIL device with a different one,
the system may misinterpret data coming from the new de-
vice. Again, you must reboot HP BASIC in order for the
system to recognize the new device.

The address of a device is simply its topological order of
placement along the link. In the above diagram, device A has
address 1, B has address 2, and C has address 3. This is only

a result of their physical order of connection. If device C had been connected between devices A and B, device A would still be address 1, but device C would be address 2 and device B address 3. The type of device has no bearing on the address assigned to it.

After the link is operational and subsequent link operations, each device looks at the data being sent down the link. If a device sees that the destination address associated with the data is the same as its address, that device receives and acts on the data. Otherwise, the data is sent on to the next device.

Preview of HP-HIL Devices

HP-HIL devices can be divided into a number of different categories. This section provides you with a table that includes these categories as well as a list of high level and low level statements that apply to each category.

HP-HIL Device Categories	High Level BASIC Access	Low Level BASIC Access
HP-HIL Keyboard	Operating system normally handles keystrokes. Programs can enter text and numbers with the IN-PUT, LINPUT, and ENTER statements.	ON/OFF KEY ON/OFF KBD KBD$
Relative Positioner (mouse, etc.)	Operating system handles as cursor movement input. Can also be used with GRAPHICS INPUT IS.	ON/OFF KBD (traps movement as arrow keys and also traps mouse buttons.) KBD$ ON/OFF KNOB ON/OFF CDIAL CDIAL
Absolute Positioner (digitizing tablet, etc.)	Can be used with GRAPHICS INPUT IS.	HIL SEND ON HIL EXT HILBUF$
ID Module	One can be used with SYSTEM$("SERIAL NUMBER").	HIL SEND ON HIL EXT HILBUF$
Other Devices	None	HIL SEND ON HIL EXT HILBUF$

Communicating Through the HP-HIL Interface

This section provides a brief description of the HP-HIL Interface Driver. This driver supports a set of statements which allow communications between the HP-HIL interface and the HP-HIL devices connected to it. Refer to the appropriate command listing in the *BASIC 5.0 Language Reference* for detailed information on these statements.

HIL SEND
Address;HIL_Command

Allows you to send HP-HIL commands to an HP-HIL device (for example, HIL SEND 1;IDD). The basic HP-HIL commands are presented in the next section. *Address* is the location of the device in the HP-HIL link. Address 1 is assigned to the first device on the link that is addressable. Subsequent addresses are assigned in ascending order.

ON HIL EXT *Address_mask*
Branch

Enables end-of-line interrupts from HP-HIL devices, allowing you to receive interrupts from up to seven devices on the HP-HIL link. *Address_mask* is a bit-map of the locations of the device or devices in the HP—HIL link. The default value is 254 which allows up to seven devices to send interrupts. *Branch* refers to a branch to a program line number, label, subroutine, or subprogram using the keywords GOTO, GOSUB, RECOVER or CALL.

OFF HIL EXT

This statement disables all previously enabled end-of-line interrupts for HP-HIL devices. Note that this statement does not require an address mask.

HILBUF$

This is a function used to capture data returned from HP-HIL devices. This function provides a 256 byte buffer for data to be stored in after execution of the first two statements listed above. Once the limit of 256 bytes has been reached, the buffer will not receive any new data until it has been emptied by a read. The first byte stored in the buffer tells you how many bytes of data have been lost. This byte is initially null.

Supported HP-HIL Devices

This section provides a brief description of those devices supported by the HP-HIL Interface Driver, and the use of a program for identifying all devices on the HP-HIL link.

Identifying All Devices on the HP-HIL Link

Each device in the HP-HIL link has a device ID that identifies the device and a Describe Record that provides you with device characteristics. This information can be obtained by executing the HP-HIL command IDD and parsing the string value returned by using the HILBUF$ function. A program called HIL_ID on the Manual Examples disc makes use of the IDD command and the HILBUF$ function for the following:

■ Determining if a device is recognized as being on the HP-HIL link.

■ Identifying the device at a specific address.

■ Determining the device's characteristics.

Assume that your HP-HIL link has the following devices:

■ Touchscreen located at address 1.

■ ITF keyboard located at address 2.

■ Function box located at address 3.

Executing the HIL_ID program will produce the following output:

```
HP 25273A (Touchscreen) located at address 1
 Describe Record Information
  I/O Descriptor Information
    Does not support Prompts/Acknowledges 1 thru 7
    Supports Proximity Detection
    Does not report buttons
  X and Y axis information reported
    Absolute positioning device
    Returns 8 bits/axis

HP 46020/21A (ITF Keyboard) located at address 2
 Describe Record Information
  No special features

HP 46086A (Function Box) located at address 3
 Describe Record Information
  I/O Descriptor Information
    Recognizes General Prompt and Acknowledge
    Does not support Prompts and Acknowledges 1 thru 7
    Does not report buttons
  No axis information reported

NO MORE DEVICES
```

The first device is a touchscreen located at address 1 in the HP-HIL link. The Describe Record provides you with the characteristics of the device. This information is as follows:

- I/O Descriptor Byte information is reported. The information supplied in this byte tells you that when you touch your finger on the screen or remove it from the screen, it will be detected. This is called proximity in/out detection.

- It is an absolute positioning device. This means that every coordinate position on the screen is referenced to the lower left-hand corner of the screen (X coordinate = 0 and Y coordinate = 0).

- X and Y axis information is reported. This tells you that Poll Records received when communicating with this device will contain X and Y coordinate information. These are absolute coordinate positions.

- Coordinate information is returned as 8 bits per axis. This means that there will be only one byte of information for each coordinate (X and Y) returned in the poll record.

HP-HIL Keyboards

There are three HP-HIL keyboards supported as HP-HIL devices on the HP-HIL link. They are:

- HP 46020/21A
- HP 98203C
- Integral keyboard
- Vectra PC keyboard

To perform interrupt branching with the keyboard keys, you need to use the following statements and function:

ON/OFF KEY

ON KEY defines and enables an event-initiated branch to be taken when a soft key is pressed. OFF KEY cancels event-initiated branches previously defined and enabled by and ON KEY statement.

ON/OFF KBD

ON KBD defines and enables an event-initiated branch to be taken when a key is pressed. OFF KBD cancels event-initiated branches previously defined and enabled by the ON KBD statement.

KBD$

This function returns the contents of the keyboard buffer when ON KBD is active.

Relative Positioners

The following devices are considered to be relative positioners:

- HP 46060A Mouse.
- HP 46083A Rotary Control Knob.
- HP 98203C Keyboard.
- HP 46094A HP-HIL Quadrature Port using HP 46095A Quadrature 3-button Mouse.

These devices support the ON/OFF KBD and KBD$ statements and functions in the same manner as described in the previous section. In addition, the following statements are also supported.

ON/OFF KNOB

ON KNOB defines and enables an event-initiated branch to be taken when the relative positioner is moved. OFF KNOB cancels event-initiated branches previously defined and enabled by the ON KNOB statement. Subsequent use of the relative positioner results in normal scrolling or cursor movement.

DIGITIZE

This statement is used when graphics input has been specified as a relative positioner by the statement

GRAPHICS INPUT IS KBD, "KBD"

It inputs the X and Y coordinates of a digitized point from the locator specified by the GRAPHICS INPUT IS statement (KBD in this case).

READ LOCATOR

This statement is used when graphics input has been specified as a relative positioner by the statement

GRAPHICS INPUT IS KBD, "KBD"

It samples the locator device without waiting for a digitizing operation.

In addition, the HP 46085A Control Dials is a device with nine knobs. Refer to ON CDIAL, OFF CDIAL, and CDIAL (n) in the *BASIC 5.0 Language Reference* manuals for information on accessing this device.

Absolute Positioners

The following devices are considered absolute positioners:

- HP 35723A HP-HIL Touchscreen
- HP 46087A A-Size Digitizer
- HP 46088A B-Size Digitizer

These devices can generate ON HIL EXT interrupts any time *except* when the absolute positioner has been specified as the input graphics device in a GRAPHICS INPUT IS statement:

```
GRAPHICS INPUT IS KBD,"TABLET"
```

Using HIL SEND to transmit a command other than IDD to these devices in this situation will result in an error. Due to the speed with which data is returned from the digitizers, an HP BASIC program cannot keep up with them using ON HIL EXT because HILBUF$ overflows. The only device in this group capable of using the ON HIL EXT statement is the touchscreen.

When these devices are specified as the graphics input device (as above in the GRAPHICS INPUT IS statement), the statements you may use are:

```
DIGITIZE X_coord,
Y_coord
```
Inputs the X and Y coordinates of a digitized point.

```
READ LOCATOR
X_coord,Y_coord
```
Samples the locator device without waiting for a digitize operation.

Security Device

The HP 46084A HP-HIL Module is an HP-HIL device that returns an identification number that identifies you as the computer user. The identification number is unique to your particular ID module. This allows application programs to use the the ID module to control access to program functions, data bases and networks.

Other Devices

The following devices can generate ON HIL EXT interrupts and respond to various HIL SEND commands.

HP 46086A Function Box. The HP 46086A Function Box provides 32 keys to select software-defined functions. It has an LED that acts as a visual prompt for any purpose you assign to it. The HP 46086A Function Box responds to the following HP-HIL commands when sent by the HIL SEND statement:

- PRM
- ACK
- DKA
- EKA 1
- EKA 2

HP 92916A Bar Code Reader. The HP 92916A Bar Code Reader reads all standard bar codes using a wand as the input device. It provides you with an effective and reliable alternative to the time-consuming keyboard for data entry. Note that HP BASIC supports this device in both the ASCII transmit mode (where the input from the device is ASCII characters), and in the keyboard mode* where it transmits the same keycodes as an HP 46020/21A Keyboard.

When the HP 92916A Bar Code Reader is in the ASCII transmit mode, use the following statement:

```
ON HIL EXT
```

When it is in the keyboard mode, use the following statements:

```
ON KBD
ENTER KBD
INPUT
LINPUT
```

---

* When in the keyboard mode, this device returns an HP-HIL ID in the same range as an HP 46020/21A Keyboard.

6

Using SRM

Contents

6 Using SRM

Introduction

This chapter describes the use of your BASIC workstation with a Shared Resource Management (SRM) system. To use SRM, you must accomplish the following:

- Install the HP 50963A SRM Interface Card.
- Connect the SRM Interface Card to the Language Processor Card.
- Load the DCOMM and SRM binaries into your HP BASIC system.

You must have the HP 50963A SRM Interface Card installed and connected to the language processor card, and the SRM and DCOMM binaries loaded to use SRM. The SRM Interface Card must have its node address set to a unique number given to you by the SRM system manager. The default (factory set) select code for the SRM interface card is 21. Each interface card attached to the Language Processor Card *must* have a unique select code. Refer to the HP 50963A *SRM Coax Interface Installation Instructions* that was shipped with your SRM card for more information.

The chapter will cover four major areas:

- The "System Concepts" section is an overview to help you understand how the SRM system works.
- The "Using Your BASIC Workstation on SRM" section demonstrates some common operations involving shared resources.
- The "Modifying Existing Programs" section discusses ways to change BASIC programs to make them work with SRM.
- The "In Case of Difficulty" section introduces troubleshooting techniques and defines the status register contents.

System Concepts

This section explains some of the concepts of the SRM system, including descriptions of the following topics:

- An SRM network.
- Support of the BASIC language on SRM.
- Management of shared peripherals.
- Booting from SRM.
- SRM's hierarchical directory structure.
- Creating directories and files.
- Shared access to directories and files (including file locking and password protection).

What Is an SRM Network?

An SRM network is a network of individual workstations connected by coaxial cable to an SRM server. The server is connected to disc drives, printers, and plotters that all workstations share. Each workstation can access data on a central database and send files to the printers and plotters, however the workstations cannot communicate with each other. The illustration shows a typical SRM installation.
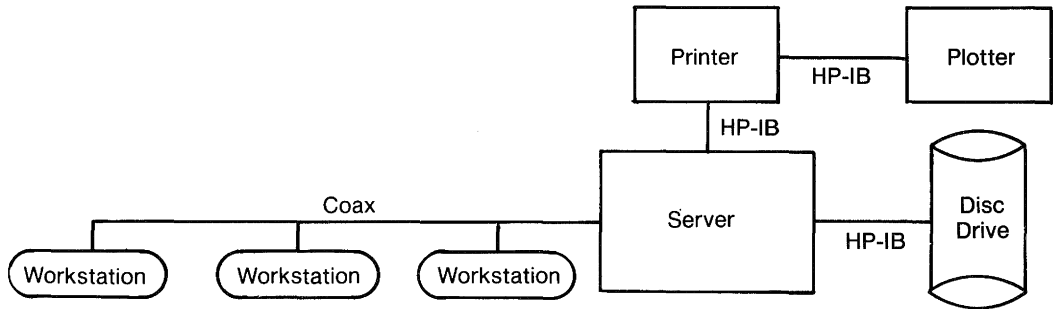
Shared Resource Support of the BASIC Language

You can use most BASIC statements that access local mass storage devices to access shared mass storage devices on SRM.

SRM adds three new commands to the BASIC mass storage statements—CREATE DIR, LOCK, and UNLOCK—and adds the PROTECT option for use with the CAT statement. In addition, the PROTECT statement's use on SRM is distinct from its use with local files.

How the SRM System Manages Shared Peripheral Use

The SRM system not only provides shared access to printers and plotters, but also manages their use so that workstations never need to wait for output to be generated.

To use shared peripherals, you place files to be output into a special directory where they are held until the printer or plotter is free. The system keeps track of the order in which files arrive from the workstations, and outputs them in the same order. This method is called "spooling," and the directory where the files are kept is called the "spooler directory." Spooler directories are created for the SRM server's use when the shared peripherals are installed on the SRM system.

After a file is placed in a spooler directory, the workstation is free to do other processing. Please note, however, that the SRM system manages output spooling only. You cannot send information such as status codes or locations of the corners of paper from a plotter back to the workstation.

Booting From the SRM

If your workstation has Boot ROM version 3.0 or later, you will be able to boot the BASIC language system into your workstation from the SRM. Once your workstation has been installed on the SRM system, the workstation power-up scheme your system manager has implemented on your SRM determines the exact procedure you use.

Automatic Configuration. The SYSTEMS directory contains the SRM operating system (SYSTEM_SRM) binary files and loading instructions belonging to each local workstation for automatic boot procedures. You can boot your HP BASIC Language Processor from your PC mass storage or from the SRM system disc. If you want to automatically load your HP BASIC Language Processor when you boot your HP 82300 Language Processor Card, ask the SRM system manager to "STORE" your HP BASIC Language Processor in the SYSTEMS directory on the SRM system disc.

System Loading. The SYSTEMS directory also contains a file called SYSTEM_LD. This file is the system loader file, and it tells the SRM operating system to look for a file called CONFIG_LDna. This file contains the name of the language system (HP BASIC, for example) to load into the memory of the local workstation. The suffix "na" is the node address assigned to the local workstation and set on the SRM interface card. An AUTOSTna in the SYSTEMS directory contains the binaries the local workstation with node address "na" wants to load during an automatic boot process. If there is no AUTOSTna, the default AUTOST in the root directory is used. Ask your SRM system manager to create these files for you. You will need to give him the names of the binary files you want loaded for your workstation.

If you do not want automatic loading at power-up time, press the space bar after you boot your HP BASIC system. This will display all the system files. Choose the system you want by typing the two-character identifier (typically 1B, 1E, etc.) that precedes the system file name. If you do not press the space bar (and do not have a CONFIG_LDna file), the first valid system file found by SRM is loaded.

Accessing the Shared Mass Storage Device

Your workstation accesses shared resources through the SRM server which is connected to the workstation through an HP SRM interface in the workstation. The remote (SRM) mass storage device is identified by a remote mass storage unit specifier, or "remote MSUS" (similar to the local MSUS), which gives information about the SRM connection. The remote MSUS includes the following required and optional information:

- The device type REMOTE, which specifies the SRM system.
- (Optional) The interface select code of your workstation's SRM interface. The default is the select code of the interface through which the boot ROM activates your workstation. (If you do not boot from the SRM, the default is the lowest select code of those available among the SRM interfaces in your workstation.)

- (Optional) The server's node address.
- (Optional) The volume name and volume password.

In general, the first step in accessing a mass storage device is to make that device the MASS STORAGE IS device. Type:

```
MSI ":REMOTE"  (ENTER)
```

SRM's Hierarchical Directory Structure

A directory is a file that is used to organize and control access to other files. The SRM operating system uses a hierarchical directory structure to organize and control access to files on a shared mass storage device just as MS-DOS uses the hierarchical directory structure for its mass storage.

Directories are a type of file and, as such, can be:

- Created with the CREATE DIR statement. When a directory is created, its location in the hierarchical structure is fixed.
- Cataloged with the CAT statement, renamed with the RE-NAME statement, and protected with the PROTECT statement.
- Filled with subordinate files and directories using the COPY, CREATE BDAT, CREATE ASCII, CREATE DIR, SAVE, STORE, RENAME, RE-SAVE, and RE-STORE statements. Each subordinate file or directory is described in its superior directory.
- Opened and closed with the MASS STORAGE IS (MSI) statement. When a user's MSI statement specifies a directory, any previously opened directory of that user is closed and the new one is opened.
- Emptied by removing all subordinate files and directories with the PURGE statement.
- Purged with the PURGE statement. You must empty and close a directory before purging it.

Using Your BASIC Workstation on SRM

This section describes, through examples, some of the more common procedures you'll use when operating your BASIC workstation on the SRM, including:

- Accessing the shared mass storage device.
- Creating directories and files.
- Listing a directory's contents.
- Shared access to remote directories and files.
- Protecting files and directories.
- Passwords and protect codes.
- Copying files.
- Purging remote files and directories.
- Using a shared printer or plotter.
- Returning to local mass storage.

For the following examples, assume you are working with the directory structure shown in the illustration below.
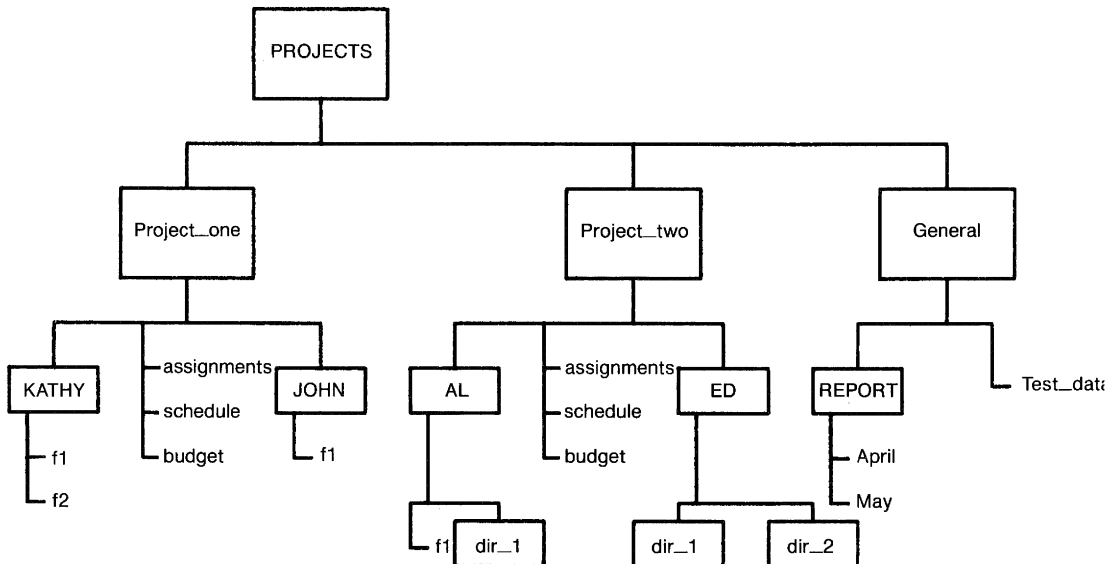
Accessing the Shared Mass Storage Device

Referring to Directories and Files in the Hierarchy. To access either a directory or a file, you must specify its location in the hierarchical directory structure. This location is specified by a list of directories, called a directory path, that you must follow to reach the desired file or directory. Directory names in the list are delimited by a slash (/).

For example, in the directory structure illustrated previously, the remote file specifier:

```
"/PROJECTS/Project_one/JOHN/f1"
```

defines the "path" to the file, *f1*, through its superior directories.

The path to a file begins either at the root level or at the current working directory. The working directory is the directory specified by the most recent MASS STORAGE IS statement.

Creating Directories. To create a directory named *CHARLIE* in the directory, *Project_one, you could type:*

```
MSI ":REMOTE"  (ENTER)
CREATE DIR "/PROJECTS/Project_one/CHARLIE"  (ENTER)
```

The leading slash indicates that the directory path begins at the root of the SRM directory structure.

You could accomplish the same thing by typing:

```
CREATE DIR "PROJECTS/Project_one/CHARLIE:REMOTE"  (ENTER)
```

Using the leading slash to begin the directory path at the root works only if you have previously established the remote mass storage as your workstation's mass storage (with some form of the MSI ":REMOTE" statement).

This statement would place your newly-created directory into the directory structure as shown below.

```
                    ┌─────────────┐
                    │  PROJECTS   │
                    └──────┬──────┘
          ┌────────────────┼──────────────────────┐
   ┌──────┴──────┐              ┌──────────────┐  ┌──────────┐
   │ Project_one │              │ Project_two  │  │ General  │
   └──────┬──────┘              └──────────────┘  └──────────┘
   ┌──────┼──────┐
┌──┴──┐ ┌─┴──┐ ┌─┴─────┐
│KATHY│ │JOHN│ │CHARLIE│
└─────┘ └────┘ └───────┘
```
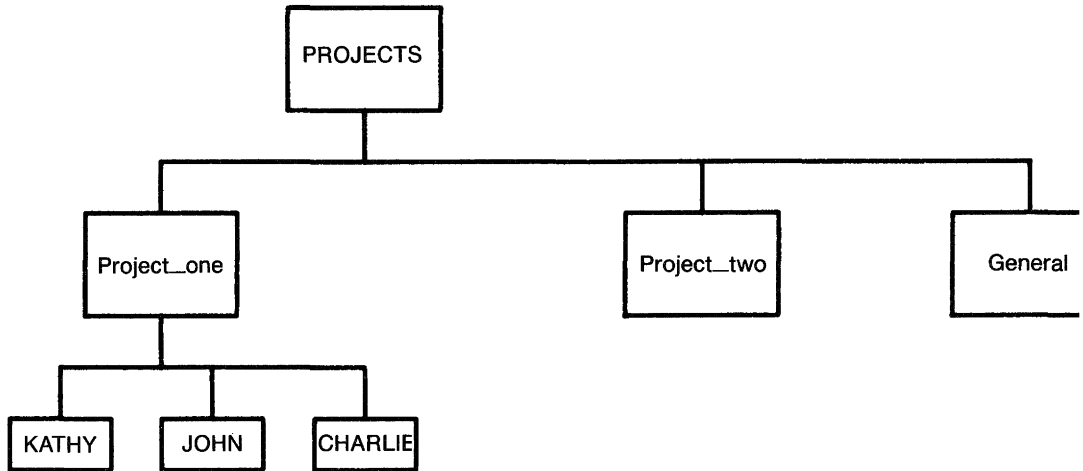
Creating Files and Other Directories Under a Directory. To create files subordinate to a new directory, you may either establish the new directory as the working directory or specify the directory path to that directory. Assuming your current working directory is the root, you could type:

```
MSI "PROJECTS/Project_one/CHARLIE"  (Enter)
```

to move into the directory, *CHARLIE*.

You could verify the new working directory with a catalog listing by typing:

```
CAT  (Enter)
```

On a computer whose screen supports an 80-character line width, the resulting listing would look like this:

```
PROJECTS/Project_one/CHARLIE:REMOTE 21, 0
LABEL:     Disc1
FORMAT:    SDF
AVAILABLE SPACE:      54096
                SYS  FILE    NUMBER    RECORD      MODIFIED    PUB OPEN
FILE NAME   LEV TYPE TYPE   RECORDS   LENGTH DATE     TIME    ACC STAT
=========== === ==== ====   =======   ====== ==== =======    === ====
```

To create an ASCII file named *ASCII_1* that is initially to contain 100 records and be contained within *CHARLIE*, you would type:

```
CREATE ASCII "ASCII_1",100   (Enter)
```

To create another directory called *MEMOS* within *CHARLIE*, you would type:

```
CREATE DIR "MEMOS"   (Enter)
```

The additions would make the directory structure look like this:
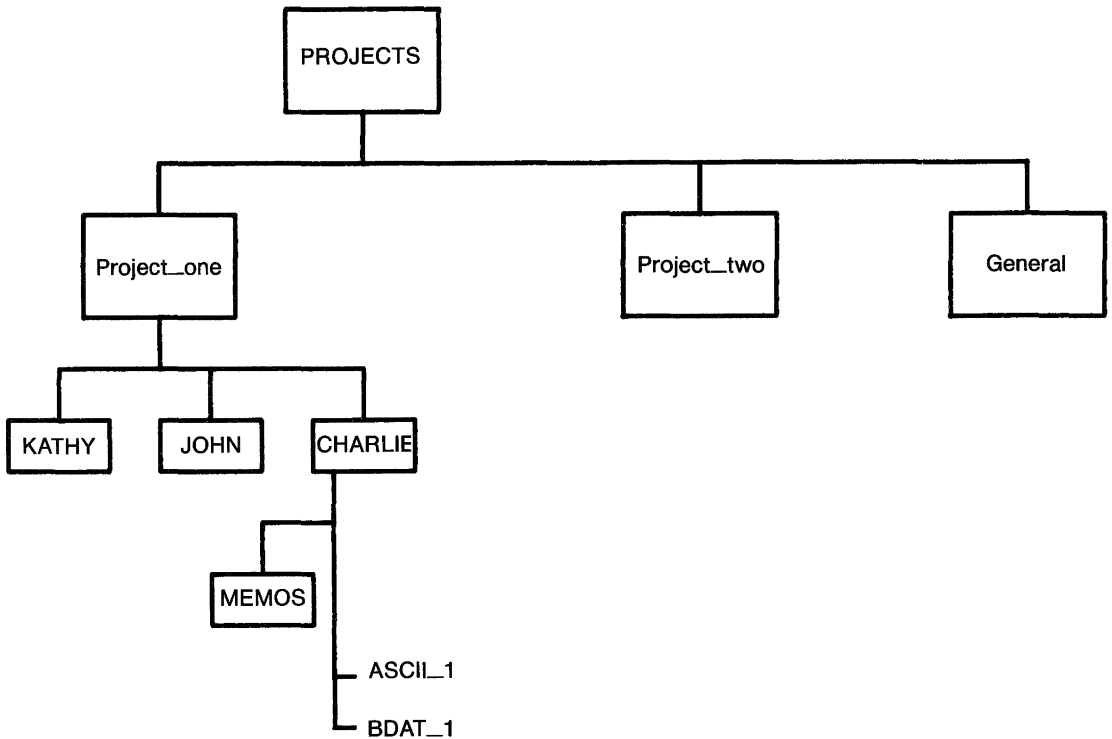
The simplest form of the CAT statement:

```
CAT   (Enter)
```

lists the contents of the current working directory because no directory is specifically identified. If no directory name is shown in the directory header, the current working directory is the root.

If you want to list the contents of *CHARLIE,* but your current working directory is *not CHARLIE,* you could:

- Designate *CHARLIE* as the working directory with the MSI statement, then use the CAT statement's "short form." For example:

```
MSI "PROJECTS/Project_one/CHARLIE:REMOTE"  (Enter)
CAT  (Enter)
```

- In the CAT statement, specify the entire path to *CHARLIE,* starting at the root, by beginning the path name with a slash(/). For example:

```
CAT "/PROJECTS/Project_one/CHARLIE"  (Enter)
```

- This form assumes that you have already designated remote mass storage with some form of the MSI ":REMOTE" statement. If you have not, use the form:

```
CAT "PROJECTS/Project_one/CHARLIE:REMOTE"  (Enter)
```

- The leading slash is not necessary, because including :REMOTE specifies the root as the beginning of the path.
- If you were in *MEMOS* (the directory immediately subordinate to *CHARLIE,*) you could use the ".." notation. For example:

```
CAT ".."  (Enter)
```

Shared Access to Remote Directories and Files

Because the sharing of files is a consequence of shared mass storage, the SRM system provides features for controlling access to shared information.

The SRM system offers three kinds of access capability for files and directories: READ, WRITE, and MANAGER. Capabilities are either public (available to all workstations on the SRM) or protected (available only to users who know the appropriate password).

Capabilities are protected with the PROTECT statement, which associates password(s) with one or more access capabilities. One password can be used to protect one or more capabilities. Each file or directory can have several password/capability pairs assigned to it.

Once assigned, the password protecting an access capability must be included with the file or directory specifier to execute statements requiring that access. If you don't specify the correct password when it is required, the system will report an error and deny access to the file or directory.

READ access capability for a file allows you to execute statements that read the file. READ access capability for a directory allows you to execute statements that read the file names in the directory, and to "pass through" the directory when the directory's name is included in a directory path.

For example, in the remote file specifier

```
"/PROJECTS/Project_one<READpass>/JOHN/f1"
```

including the assigned password <READpass> allows passage through the directory *Project_one* to allow access to its subordinate directories and files.

WRITE access capability for a file permits you to execute statements that write to the file. WRITE access capability for a directory allows you to execute statements that add to or delete from the directory's contents.

With the MANAGER access capability, public capabilities for a file or directory differ slightly from password-protected capabilities. Public MANAGER capability allows any SRM user to PROTECT, PURGE or RENAME the file. The password-protected MANAGER capability provides MANAGER, READ and WRITE access capabilities to users who include a valid password in the file or directory specifier.

Protecting Files and Directories

When you create directories and files, their access capabilities are "public" (available to any user on the SRM). You may subsequently protect a directory or file against certain types of access by other SRM workstations, provided:

- you have MANAGER access capability on the file or directory (MANAGER access to the file is public or you know the password protecting the capability);

- you have READ access capability on the directory immediately superior to the file or directory you wish to protect;

- you protect the file or directory either while "in" its superior directory or by specifying the valid directory path to its superior directory.

For example, using the directory structure established for other examples in this section (see illustration) and assuming no passwords have been assigned to the files, you could:

```
                    ┌──────────────┐
                    │   PROJECTS   │
                    └──────────────┘
                            │
        ┌───────────────────┼───────────────────────────┐
┌──────────────┐    ┌──────────────┐            ┌──────────────┐
│  Project_one │    │  Project_two │            │   General    │
└──────────────┘    └──────────────┘            └──────────────┘
        │                   │
  ┌─────┼─────┐       ┌─────┼──────────┐
┌──────┐┌──────┐┌──────────┐  ┌──────┐   ┌──────────────┐  ┌──────┐
│KATHY ││ JOHN ││ CHARLIE  │  │  AL  │   └ Assignments  │  │  ED  │
└──────┘└──────┘└──────────┘  └──────┘   └──────────────┘  └──────┘
                      │           │
              ┌───────┘      ┌────┴─────┐
        ┌──────────┐              ┌──────────┐
        │  MEMOS   │              │  dir_1   │
        └──────────┘              └──────────┘
              ├ ASCII_1       ├ f1
              │               └ BDAT_1
              ├ BDAT_1
              └ Test_prog
```
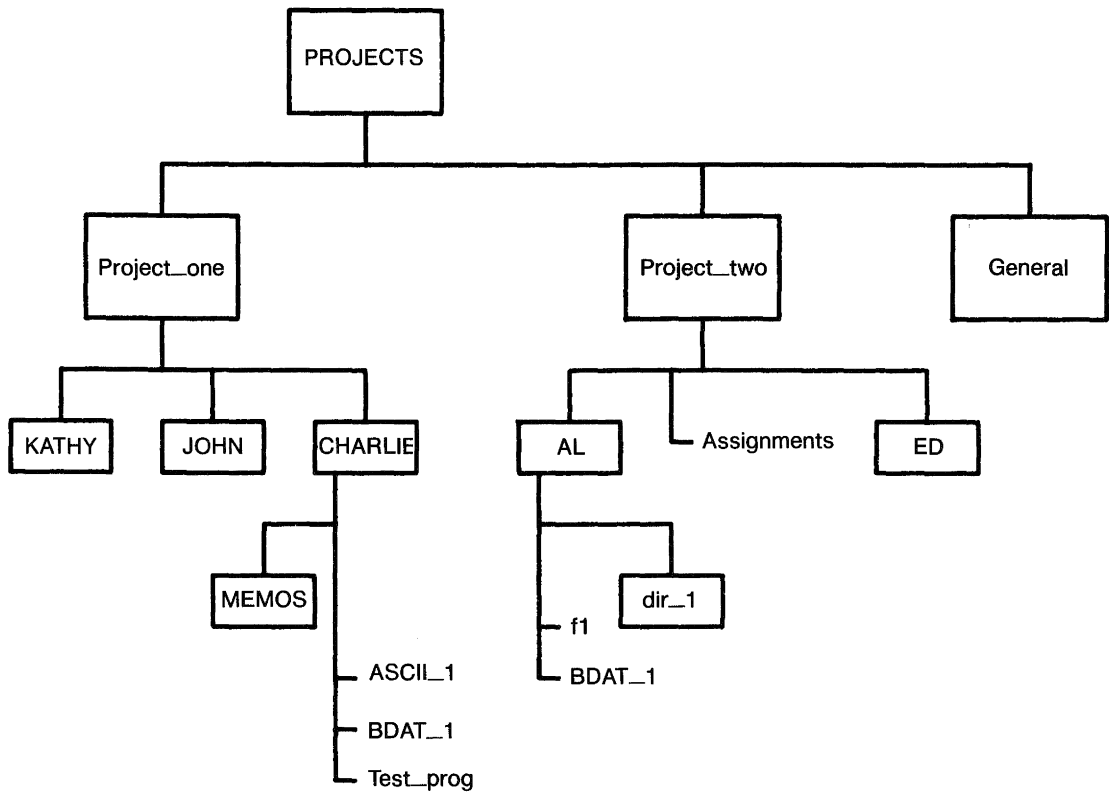
1. Assign the password *passme* to protect the MANAGER
and WRITE access capabilities on the directory *CHARLIE*
with the sequence:

```
MSI"/PROJECTS/Project_one"  (Enter)
PROTECT "CHARLIE",("passme":MANAGER,WRITE)  (EXECUTE)
```

which executes the PROTECT statement after moving to
the directory *Project_one* (immediately superior to
CHARLIE). As a result of the PROTECT statement, the
READ access capability on *CHARLIE* is still public, but
any operations that require MANAGER or WRITE ca-
pabilities must include the password.

2. Remove all public access capabilities from the file *ASCII_1* by assigning the password *no_pub,* using:

```
PROTECT "CHARLIE/ASCII_1",("no_pub":MANAGER,WRITE,READ)   (Enter)
```

or

```
MSI "CHARLIE"   (Enter)
PROTECT "ASCII_1",("no_pub";MANAGER,WRITE,READ)   (Enter)
```

These statements assume you are in the directory, *Project_one,* as if you had executed the statements in the previous step.

The second sequence of statements makes *CHARLIE* the new working directory, whereas in the first, you merely "pass through" *CHARLIE* to reach *ASCII_1.* With the READ access capability on *CHARLIE* still public, you do not need a password.

3. Protect the file, *BDAT_1,* so that data can be read from it but not written into it without using the password, *write.* If the current working directory were *CHARLIE,* you would type:

```
PROTECT "BDAT_1",("write":MANAGER,WRITE)   (Enter)
```

4. Protect the MANAGER access capability of the directory *MEMOS* with the password, *mgr_pass* (so that everyone can read from and write to the directory, but a password is required to purge the directory or its contents) by typing:

```
PROTECT "MEMOS",("mgr_pass":MANAGER)   (Enter)
```

If you protected the files and directory in *CHARLIE* as in the steps above, a catalog listing of *CHARLIE* will look something like this:

```
   PROJECTS/Project_one/CHARLIE:REMOTE 21, 0
   LABEL:      Disc1
   FORMAT:     SDF
   AVAILABLE SPACE:       54096
                 SYS  FILE    NUMBER    RECORD     MODIFIED      PUB OPEN
FILE NAME     LEV TYPE  TYPE   RECORDS   LENGTH DATE    TIME     ACC STAT
============  === ====  ====   =======   ====== ==== ==========  === ====
ASCII_1        1         ASCII      0       256  2-Dec-84  13:20
BDAT_1         1 98X6   BDAT       0       256  2-Dec-84  13:20   R
MEMOS          1         DIR        0        24  2-Dec-84  13:20   RW
```

The letters in the column labeled PUB ACC indicate access capabilities that are public (not protected with a password). For example, only the MANAGER ⟨M⟩ access capability on the directory *MEMOS* has been protected, leaving the READ ⟨R⟩ and WRITE⟨W⟩ capabilities available to any SRM workstation user.

Specifying Passwords. When a password is required, you must include the correct password as part of the file or directory specifier in any command or statement that requires the protected access on the file or directory. The password must be enclosed between "<" and ">", and must immediately follow the name of the file or directory it protects.

For example, to get the file *ASCII_1*, you might type:

```
GET "/PROJECTS/Project_one/CHARLIE/ASCII_1<no_pub>"   (Enter)
```

If the password were not included in the specifier, the system would respond with an error message and refuse to get the file.

Exclusive Access: Locking Files. Although sharing files saves disc space, allowing several users access to one copy of a file introduces the danger of users trying to access the file at the same time, which can cause unpredictable results. For instance, if one user tries to read part of a file while another user is writing to it, the file's contents may be inaccurate for the read.

You can "lock" a shared file with the LOCK statement, giving you sole access to that file. The same file can be locked several times in succession. Unlocking a file requires that you cancel all locks on that file. If you use the UNLOCK statement, you must cancel each LOCK with a corresponding UNLOCK. Using ASSIGN to re-open a locked file unlocks the file and you must execute another LOCK statement to lock the file again. Closing the file via ASSIGN @...TO * cancels all locks on the file.

In this example, a critical operation must be performed on the file named *File_a*, and you do not want other users accessing the file during that operation. The program might be as follows:

```
1000    ASSIGN @File TO "File_a:REMOTE"
1010    LOCK @File;CONDITIONAL Result_code
1020    IF Result_code THEN GOTO 1010    ! Try again
1030    !  Begin critical process
   .
   .
   .
2000    !  End critical process
2010    UNLOCK @ File
```

The numeric variable called *Result_code* is used to determine the result of the LOCK operation. If the LOCK operation is successful, the variable contains 0. If the LOCK is not successful, the variable contains the numeric error code generated by attempting to lock the file.

Passwords and Protect Codes

The PROTECT statement format for remote files is different from the format for local files. Depending on the type of mass storage is being used, you can use either of the following to decide which syntax will be used:

1. Try the non-SRM syntax with an ON ERROR statement enabled. If an error occurs, see if it indicates that the mass storage device is an SRM. An Error 1 occurs when the following statement is executed on a remote file.

PROTECT   *file specifier,protect code*

2. If the program uses a string to store the mass storage unit specifier, check for a non-zero value of POS (Msus$,"REMOTE"). This alternative is easier to implement than alternative 1 but will not work if the program accesses the default device when Msus$ is empty.

If the program looks for a password error (Error 62) at ASSIGN time, the program may have to be modified because the system may not detect the password error until an ENTER @Path or OUTPUT @Path is attempted.

Copying Files

With SRM, you can copy files between local and remote mass storage devices by any of the methods illustrated in the following examples. Again using the directory structure established for the other examples in this section, assume that the current working directory is *CHARLIE*.

Using the COPY Statement. The most direct method of copying a file from local to remote mass storage is to use the COPY statement. For example, to copy a PROG file named *Test_prog* that is on a local disc drive into the directory *CHARLIE* on the SRM system disc, you could type:

```
COPY "Test_prog:,1500,0" TO "Test_prog"   (Enter)
```

By including the :,1500,0 MSUS, you can access the local mass storage without changing the current working directory (which is a remote directory).

Caution	Do not copy entire volumes between built-in disc drives and SRM drives. Copy each file separately.
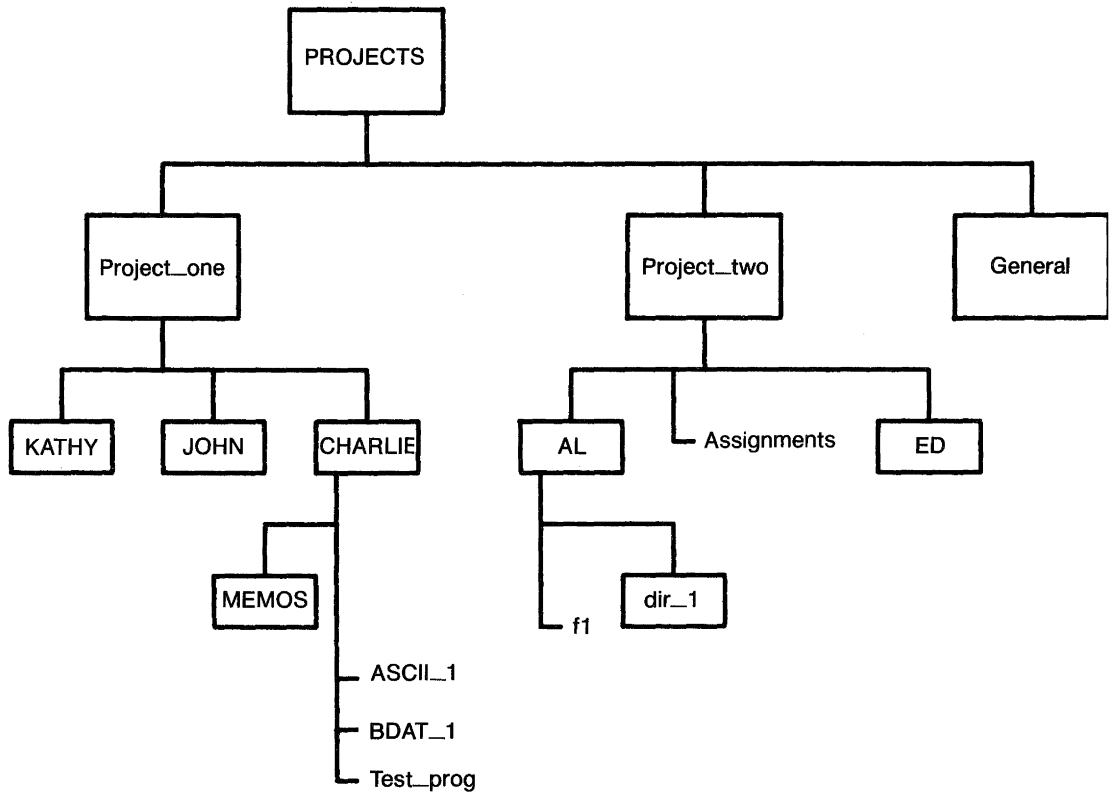
Other Uses of COPY. The COPY statement can be used to copy files not only from local to remote mass storage but also from remote to local mass storage and from one remote mass storage device to another. However, you cannot copy an entire remote mass storage volume in a single COPY statement. (You must copy a remote volume file by file.)

Suppose you want to copy the file *BDAT_1* from the directory *CHARLIE* into the directory *AL* (see previous illustration).

Assuming the working directory is *CHARLIE,* you could type:

```
COPY "BDAT_1" TO "/PROJECTS/Project_two/AL/BDAT_1"  (Enter)
```

The effect of the copy on the directory structure is illustrated below:
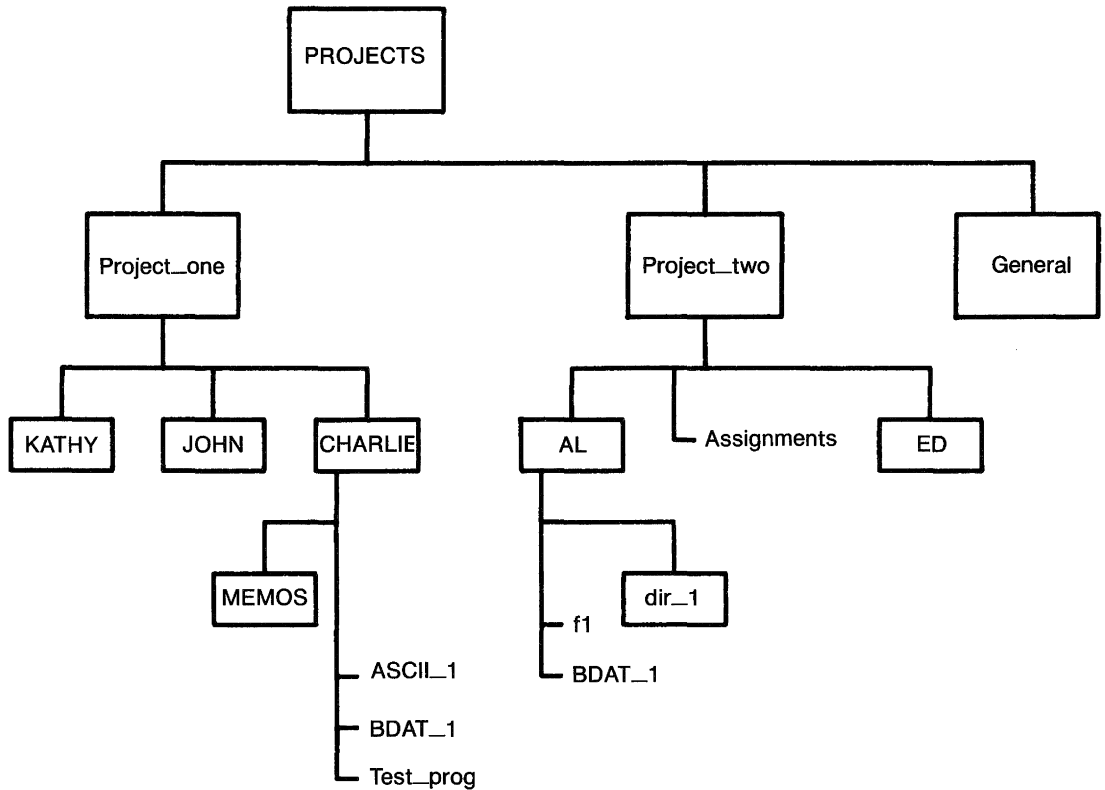
Using LOAD and STORE. You may also copy files by loading the program into your workstation from local mass storage and then storing it in remote mass storage. For example, to copy a PROG file named *Test_prog* that is on a disc in your workstation's A: disc drive into the directory *CHARLIE* on the SRM system disc (as demonstrated earlier using COPY), you could type:

```
LOAD "Test_prog:,1500,0"  (Enter)
```

Once the file is in your workstation's memory, you may then store the file in the remote directory by using a statement such as:

```
STORE "Test_prog:REMOTE"  (Enter)
```

Copying Item-by-Item Using ENTER and OUTPUT. You may also copy a file from local to remote mass storage an item at a time, as illustrated in the programs that follow. These programs use the ENTER and OUTPUT statements to copy data item-by-item from a local BDAT file to remote mass storage.

The first program creates and fills a BDAT file named *BDAT_FILE*.

```
10   CREATE BDAT "BDAT_FILE:,1500,0",10
20   ASSIGN @Local TO "BDAT_FILE:,1500,0"
30   !
40   FOR Item=1 TO 50
50   OUTPUT @Local;"String data item"
60   NEXT Item
70   !
80   ASSIGN @Local TO *
90   END
```

The second program copies the contents of *BDAT_FILE* item-by-item into a file (also called *BDAT_FILE)* in the SRM directory named *General* (shown in the previous illustration).

```
100   DIM String_item$[20]
110   CREATE BDAT "PROJECTS/General/BDAT_FILE:REMOTE",10
120   ASSIGN @Local TO "BDAT_FILE:,1500,0"
130   ASSIGN @Remote TO "PROJECTS/General/BDAT_FILE:REMOTE"
140   !
150   FOR Item=1 TO 50
160   ENTER @Local;String_item$
170   OUTPUT @Remote;String_item$'
180   NEXT Item
190   !
200   ASSIGN @Local TO *
210   ASSIGN @Remote TO *
220   END
```

Purging Remote Files and Directories

The PURGE statement works the same for removing remote files as for removing files from local mass storage. You may also remove directories using PURGE. PURGE works only with closed files and directories. Directories must also be empty (not contain any files or directories). Refer to the discussion on "Returning to Local Mass Storage" later in this section for details on closing files and directories.

When specifying the remote file to be purged, you must include all passwords protecting access capabilities required for the PURGE. For example, to purge the file *BDAT_1* from the directory *CHARLIE* (see previous examples), you could type:

```
PURGE ".<passme>/BDAT_1<write>"   (Enter)
```

In this example, *CHARLIE* is the current working directory, as denoted in the directory path by ".".

To purge a file, you must have the MANAGER access capability on that file and READ and WRITE access capabilities on the file's superior directory. Because *passme* protects the WRITE capability on *CHARLIE* and *write* protects the MANAGER capability on *BDAT_1*, both passwords must be included in the file specifier in the PURGE statement.

Although you do not normally need to specify the working directory in a directory path, you must include the password for the PURGE operation. The READ capability on *CHARLIE* is not password-protected.

To purge *CHARLIE*, you would first need to purge the remaining files and directory in *CHARLIE*. Because the MSI statement "opens" a directory (making it the current working directory), you must also "close" *CHARLIE*.

For example, if no files or directories remained in *CHARLIE*, you could purge *CHARLIE* by typing.

```
MSI ":REMOTE"  (Enter)
PURGE "PROJECTS/Project_one/CHARLIE<passme>  (Enter)
```

The first statement closes *CHARLIE* and establishes the root directory as the current working directory. Note that, because *passme* protects the MANAGER access capability on *CHARLIE*, you must include that password in the PURGE statement.

Using a Shared Printer or Plotter

Use of special SRM directories called "spooler directories" allows you to access a shared printer or plotter. Setting up a spooler directory is explained in the *Shared Resource Management System Manager's Guide*. The examples in this section assume that the spooler directories *LP* (for "Line Printer") and *PL* (for "PLotter") have been created at the root of the SRM directory structure.

Spooling Using PRINTER IS and PLOTTER IS. You can use the PRINTER IS and PLOTTER IS statements to send data to your shared printer or plotter. The following command sequence illustrates this spooling method:

```
CREATE BDAT "/LP/Print_file",1
PRINTER IS "/LP/Print_file"
LIST
XREF
PRINTER IS CRT
```

PRINTER IS and PLOTTER IS statements work only with BDAT files.

Note   The DUMP DEVICE IS and PRINTALL IS statements do *not* support files, so they cannot be used for printer spooling.

Writing Files to the Spooler Directories. You may also access the printer associated with *LP* by placing the data to be printed in an ASCII or BDAT file in that spooler directory. For example, to list a program currently in memory, you could SAVE the program in *LP* as the file *P1_LISTING* by typing either:

```
SAVE "LP/P1_LISTING:REMOTE"  (Enter)
```

or

```
SAVE "/LP/P1_LISTING"  (Enter)
```

The SAVE statement creates an ASCII file. Although this is the same syntax used to save programs on a shared disc, the SRM system knows that *LP* is a spooler directory and prints the file as soon as possible.

When used for spooling, SAVE places a file in the spooler directory. The file is printed, then purged. You may wish to save or create the file first, then use the COPY statement to place the file into the spooler directory.

Sending Program Output to a Shared Printer. To spool program output to a shared printer, create an ASCII or BDAT file, assign an I/O path name to the file (which opens the file), and OUTPUT the data to that file. With BDAT files, you should ASSIGN with FORMAT ON. When the file's contents are to be printed, close the file. The following example program segment outputs the data stored in the string array called *Data$* to an ASCII file named *PERFORMANCE.*

```
760    CREATE ASCII "/LP/PERFORMANCE",100
770    ASSIGN @Spool TO "/LP/PERFORMANCE"
780    OUTPUT @Spool;"Performance Summary"
790    OUTPUT @Spool;Data$(*)
800    ASSIGN @Spool TO *     ! Initiate printing.
```

The system waits until the file is not empty and closed before sending its contents to the output device. If your file is not printed or plotted within a reasonable amount of time, you may not have closed it. You can verify that your file is ready to be printed or plotted by cataloging the spooler directory:

```
CAT "/LP"  (Enter)
```

The open status (OPEN   STAT) of the file currently being printed or plotted is listed as locked (LOCK). Files currently being written to the spooler directory (either printer or plotter) are listed as OPEN. Files that do not have a status word in the catalog are ready for printing or plotting.

The SRM 2.0 and newer operating systems allow BDAT files to be sent to the printing device as a byte stream.

Note	With the SRM 2.0 and newer operating systems, a BDAT file sent to the spooler is printed exactly as the byte stream sent. Unless you set up the BDAT file correctly, improper printer output or operation could result. Therefore, you should ASSIGN BDAT files with FORMAT ON before outputting data.

The spooler inserts a carriage return and line feed after each record in an ASCII file. To put several strings on one line, concatenate them into one string before using OUTPUT to send them to the spooler file. You may insert ASCII control characters in the data by using the CHR$ string function.

Appearance of Output. Printed output for each file includes a one-page header, which identifies the directory path to the file, the file's name, and the date and time of the printing. You can disable this header by using the NOBANNER option at the server.

To cause the printer to skip the paper perforation after printing a page (60 lines), prefix your file name with "FF". For example:

```
SAVE "/LP/FF_MYTEXT"  (Enter)
```

Aborting Printing/Plotting in Progress. To abort an in-progress printing or plotting, use the SPOOLER ABORT command from the SRM server. The system stops sending data to the output device and closes and purges the file. For details on bringing the spooler UP and DOWN, see the description of the SPOOLER command in the "Language Reference" section of the SRM *System Manager's Guide*.

With SRM 2.0 and newer operating systems, if a printer is taken off-line while a file is being printed, the printer stops and resumes when the printer is put back on-line. No data is lost during such an interruption. The SRM 1.0 operating system also resumes printing, but from the beginning of the file.

Returning to Local Mass Storage

When you have finished accessing shared resources, you should close all of your files and directories to "release" the system resources.

Remote files are closed by ASSIGN ... TO (*). The SCRATCH A command closes directories and files. All remote files except those opened with the PRINTER IS statement are also closed by pressing RESET.

To close your current working directory, execute an MSI to a local msus. For example,

```
MSI ":,1500,2"
```

If you booted from local mass storage, you may also execute the SCRATCH A command to completely release your access to the system. If you booted from the SRM, executing SCRATCH A resets the current working directory to the root.

Modifying Existing Programs to Access Shared Resources

This section summarizes ways you can modify existing programs that access local resources to allow those programs to access shared resources.

When modifying programs to access SRM mass storage device(s), you should be aware that:

■ Local and remote mass storage file specifiers may differ and string variable names that contain file specifiers may need corresponding modification.

- References to mass storage unit specifiers throughout the program may have to be altered.
- Allowances may have to be made for directory path specification.
- Local protect codes may differ from passwords on remote files. The syntax for protecting remote files is different from that used for local files.

File Specifiers

Composition of File Names. All files names for local mass storage are 1 to 10 characters long, while remote file names contain 1 to 16 characters. Remote file names can contain the period character (.), while local files cannot. If file name compatibility between resources is required, use 10 or fewer characters and do not use periods within remote file names.

File and Mass Storage Device Specification in String Variables. Modifying programs for use with shared resources generally requires changing the value, and often the length, of the string variables used to specify files and mass storage devices. The statements that assign the actual values to the string variable may have to be modified individually.

Mass Storage Unit Specification

Some programs use separate variables for the file name and MSUS. For example:

```
ASSIGN @Path TO Filename$&Msus$
```

If so, both variables may have to be dimensioned to greater lengths. Allowing 34 characters for the file name variable accommodates a 16-character file name, a 16-character password, and the "<" and ">" password delimiters (for example, "ABCDEFGHIJ123456<1234567890123456>"). The remote MSUS may occupy up to 54 characters.

Other programs may use MASS STORAGE IS statements throughout the program instead of including the MSUS in each file specifier. For instance:

```
MASS STORAGE IS Left_drive$
ASSIGN @File To File_name$
```

Unless variable(s) are used to specify the MSUS and each variable is assigned a value in only one place, you may have to modify each MASS STORAGE IS statement to specify the desired remote mass storage device.

Allowing For Directory Paths

Suppose the following program needs to be modified to include a remote file's directory path.

```
100   DIM Filename$[14],Msus$[20]
  .
  .
  .
200   Filename$="SLIDES"
210   Msus$=":HP9895,700"
  .
  .
  .
300   ASSIGN @File TO Filename$&Msus$
310   OUTPUT @File;Data(*)
320   ASSIGN @File TO *
  .
  .
  .
400   ASSIGN @File TO Filename$&Msus$
410   OUTPUT @File;Data(*)
420   ASSIGN @File to *
```

In this example, it is probably easiest to add another string variable for the (optional) path name. For example:

```
100   DIM Dir_path$[160],Filename$[80],Msus$[80]
  .
  .
  .
200   Dir_path$="FRED/DATA_FILES/"
210   Filename$="SLIDES"
220   Msus$=":REMOTE 21,1"
  .
  .
  .
300   ASSIGN @File TO Dirpath$&Filename$&Msus$
310   OUTPUT @File;Data(*)
320   ASSIGN @File TO *
```

If the Dir_path$ variable is null, the statement looks exactly like it did before the modification. If the Msus$ variable is null, the current mass storage device is accessed. The only difference is in the allowable length of the string variables.

In Case of Difficulty

If you have problems using the SRM network from your PC workstation, verify the following items:

■ The language processor card and the SRM card are properly installed and connected.

■ The language processor card is functioning correctly.

■ The node address on the SRM interface card has been set to a unique number assigned by the SRM system manager.

■ The SRM and DCOMM binaries are loaded in your HP BASIC system.

- See if you can access the SRM system disc from your PC workstation. Use the MSI ":REMOTE" statement to set your mass storage unit specifier to the SRM system disc, and then use CAT to display the SRM root directory.

- From the SRM server, type "NODES SC", where SC is the select code of the server's SRM interface card. If there is only one SRM card in the server, the select code is probably 21. The server recognizes your PC workstation if the display on the server shows the node address of your PC workstation's SRM interface card.

- Enter and run the following HP BASIC program to determine the contents of the SRM status registers:

```
10   FOR I = 1 TO 12
20     IF I = 4 THEN
30       PRINT "Register 4 not implemented."
40     ELSE
50       STATUS 21,I;Number
60       PRINT "Register ",I,"equals ",Number
70     END IF
80   NEXT I
90   END
```

- Record the results. They will help Hewlett-Packard support center personnel if you cannot find the problem.

- Run the diagnostics provided for the SRM card and record the results.

The troubleshooting section in the *SRM System Manager's Guide* contains complete instructions for diagnosing SRM network problems. Contact your SRM manager for assistance.

Summary of SRM Status Registers

Status Register 0   Card Identification

52 if the Remote Control switch (R) is set to 0 (closed); 180 if switch is set to 1 (open).

Status Register 1   Interface Interrupts

1=interrupts enabled; 0=interrupts disabled.

Status Register 2   Interface Busy
1=busy; 0=not busy.

Status Register 3   Interface Firmware ID

Always 3 (the firmware ID of the interface).

Status Register 4   Not Implemented

Status Register 5   Data Availability

0=receiver buffer empty;
1=receiver data available but no control blocks buffered;
2=receiver control blocks available but no data buffered;
3=both control blocks and data available.

Status Register 6   Node Address of the interface

Node address of the SRM interface installed in this computer which is set to the specified select code. The range of node addresses is 0 through 63.

Status Register 7   CRC Errors

Total number of cyclic redundancy check (CRC) errors detected by the interface since powerup or RESET.

Status Register 8   Buffer Overflows

Total number of times the receive buffer has overflowed since powerup or RESET.

Status Register 11   Amount of available space (number of bytes) in the transmit-data buffer.

Status Register 12   Number of transmission retries performed since powerup or RESET.

A   Keyword Reference List

This appendix has been deleted because the complete *BASIC 5.0 Language Reference* (Volumes 1 and 2) is now included in your HP BASIC system. Refer to the appropriate volume for the information you need.

B

Using the HP BASIC Language Processor in the MS-DOS Environment

The HP BASIC Language Processor uses a software emulator to simulate the I/O devices of a Series 200/300 computer. The software is installed into the language processor card. This appendix describes how the HP BASIC Language Processor affects the keyboard, display, and interface cards, and points out the differences between the HP BASIC Language Processor and Series 200/300 BASIC.

The following diagram shows the language processor card and its relationship to the host PC.
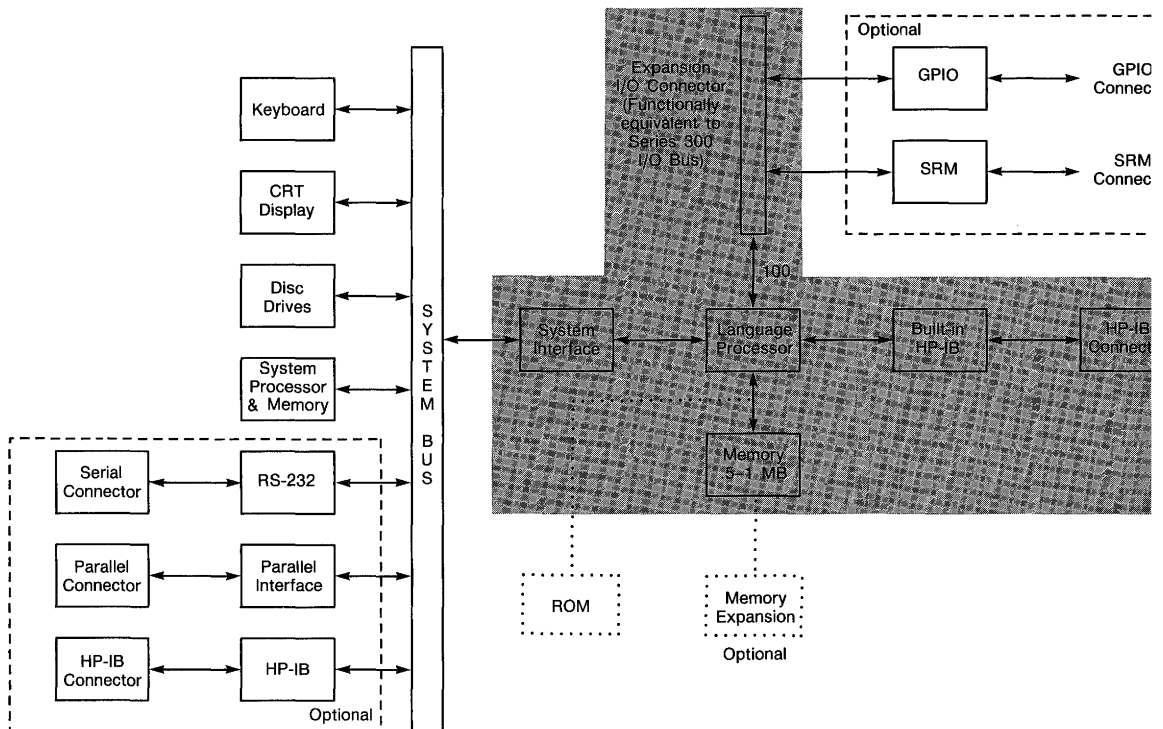
Available Devices and Interfaces

The language processor card uses computer devices as though they were standard Series 200/300 HP devices. This is completely invisible to programs running on the language processor. The knob is simulated by a mouse and the display can emulate several Series 200/300 displays.

Select code 15 corresponds to the built-in disc drives in the computer. These drives are treated as CS80 drives.

Select codes 24 and 25 correspond to HP 82990A HP-IB cards that may be plugged into the PC bus.

Select codes 9 and 23 emulate HP 98626A cards on the host PC serial interface, such as the HP 24540A serial/parallel card.

Select code 26 is used for a printer interface (LPT1) using limited GPIO capabilities in the HP BASIC language processor.

Select code 19 is used for the MS-DOS communications port. It's purpose is to allow HP BASIC language processor programs and commands to communicate with the computer's operating system. The interface between the HP BASIC language processor and the operating system is the same as if a GPIO card were installed.

READIO, WRITEIO, CONTROL, STATUS, ENTER, and OUTPUT operations are not supported for select code 15.

Select Code	Interface Type	200/300 Card Emulated	PC Bus Card
7	HP-IB	Built-in HP-IB	Built-in HP-IB
9 or 23	RS-232	HP 98626A	HP 24540A or equivalent at COM1 or COM2
12*	GPIO	HP 98622A	HP 82306A
15	HP-IB	Disc interface	Part of host PC
19	GPIO	DOS port	Part of host PC
21*	SRM	HP 50962A	HP 50963A
24	HP-IB	HP 98624A	HP 82990A
25	HP-IB	HP 98624A	HP 82990A
26	GPIO	HP 98622A (GPIO printer only)	Any system interface (i.e. a parallel printer interface) at LPT1

* Factory setting. Switches on the card may be changed to any available select code.

You must have the GPIO binary loaded in order to use select codes 19 or 26. You must have the HPIB and CS80 binaries loaded in order to use select code 15. These three binaries are included in the system shipped with your HP BASIC.

PC Bus Cards

The HP BASIC language processor supports the following:

- Up to two HP-IB PC interface cards, select codes 24 and 25.
- Up to two serial PC interface cards, select codes 9 and 23.
- One printer interface at select code 26.

HP-IB PC Cards. The two HP-IB cards will appear like HP 98624A HP-IB cards to the HP BASIC language processor. The registers are identical. All functions are available except Direct Memory Access (DMA). You can configure the card by changing the configuration file. Refer to appendix F for information on the configuration file.

The HP 82990A HP-IB card has a default bus address of 30.

Serial PC Cards. The HP BASIC language processor views serial cards the same way it views HP 98626A cards. The only function not available is DMA. You can configure the card by changing the configuration file. Refer to appendix F for information on changing the configuration file.

Note

Although the functionality of serial and HP-IB cards is equivalent to the Series 200/300, there may be a slight reduction in performance under the HP BASIC Language Processor.

Printer Interface. One printer is supported as defined by LPT1. The GPIO (PSTS) line is used to signify an error condition such as being out of paper. The card appears like an HP 98622A GPIO, although many bits in the registers are not functional.

HP-IL printers are not supported using select code 26.

The printer supported at select code 26 will respond with timeouts just as an HP-IB printer on the internal select code 7, except that the timeout unit when using select code 26 is half-minutes instead of seconds. Thus, ON TIMEOUT 26, 1 will provide a 30 second timeout, but ON TIMEOUT 7, 1 will provide a 1 second timeout.

HP BASIC uses the Roman-8 symbol set as the default for video display. You can customize the displayed font and symbol set. Refer to "Video Display" in this appendix.

SRM and GPIO Interface Cards. The HP BASIC language processor has Shared Resource Management (SRM) and General Purpose Input/Output (GPIO) interface cards that plug directly into your personal computer and provide the exact functionality of their Series 200/300 interface counterparts. The HP BASIC language processor cards are accessed exactly like the DIO counterparts.

The SRM card provides access to HP's SRM network, and the GPIO card makes it possible to connect to any user devices.

On a Series 200 computer it is possible to send a character and then execute an ENABLE INTR command. This is *not* possible on a PC. Interrupts on the PC should be enabled *before* accessing the PC card.

Video Display

The software emulator enables the HP BASIC language processor to emulate three different Series 200 display models:

- HP 9816B: black and white, 400 dots by 300 dots.
- HP 9836A: black and white, 512 dots by 390 dots.
- HP 9836C: color, 512 dots by 390 dots.

The hardware requirements for the Series 200 display models are:

- HP 9816B: Multimode, Monochrome Plus, or Enhanced Graphics Adapter (EGA) plus monitor.
- HP 9836A: Multimode, Monochrome Plus, or Enhanced Graphics Adapter (EGA) plus monitor.
- HP 9836C: EGA plus Enhanced Graphics Display or equivalent.

The display model used is determined by the configuration file, which is accessed during the boot process and remains unchanged until the configuration file is altered and the HP BASIC language processor is booted again.

If no display is specified in the configuration file, your computer will default to the HP 9816B if you are using Monochrome Plus, to the 9836A if your using Multimode, and to the 9836C if you are using EGA.

Alpha and Graphics. It is possible to choose emulation (by using the configuration file) with separate alpha/graphics planes or combined alpha/graphics emulation for each model. The default mode is combined alpha/graphics, which is the mode used in series 200/300 workstations. When using combined alpha/graphics mode, some performance is lost while displaying alpha text. For best alpha performance, select the separate alpha/graphics mode.

For faster scrolling speed use the separate alpha/graphics mode.

Display Differences. The HP 9836 display models have a resolution of 512 dots horizontally by 390 dots vertically, but your PC display may be slightly different. For example, when emulating a model 9836 display using an EGA adapter limited to 350 dots vertically, the top 40 dot-rows will not appear on the display.

The HP BASIC language processor can shrink any graphics picture to fit the display by using the VIEWPORT and WINDOW statements in your BASIC program.

Use the following command to set the soft clip limits at the boundaries of the area accessible by HP BASIC on the PC display:

```
VIEWPORT 0,RATIO*100,0,(<Pc_vert_res>/390)*100
```

RATIO is an HP BASIC function that returns the aspect ratio of the emulated display (a 9836 in this case). The value of Pc_vert_res can be derived from the HPWSTATUS B$ parameter. Refer to the section entitled "HPWSTATUS" later in this appendix.

The resultant values for the various PC displays are shown below:

- For EGA displays, use

```
VIEWPORT 0,131.3,0,89.7
```

- For Monochrome Plus displays, use

```
VIEWPORT 0,131.3,0,89.2
```

- For Multimode displays, no adjustment is necessary since there is no clipping of the display.

After you set the soft clip limits, you may wish to adjust the user display units (UDU) to obtain "square dots" so that POLYGON and RECTANGLE commands can generate circles and squares. This is necessary because the PC display does not have physically square dots like the series 200/300 display. Use the following command to adjust the UDUs:

```
WINDOW 0,<Pc_dot_ratio>*RATIO*100,0,100
```

Pc_dot_ratio is the HPWSTATUS E parameter. Refer to the section entitled "HPWSTATUS" later in this appendix.

The resultant values for the available PC displays are shown below:

- For EGA displays use

```
WINDOW 0,103.8,0,100
```

■ For Monochrome Plus displays use

```
WINDOW 0,89.3,0,100
```

■ For Multimode displays use

```
WINDOW 0,105.09,0,100
```

Note that this sets the Y axis at 100 UDUs. This can be normalized to any value by scaling the X and Y axes by the same amounts.

Refer to chapter 4, "Graphics Techniques", for more information on the VIEWPORT and WINDOW statements.

When using the combined alpha/graphics mode on EGA monitors, pens 8-15 map to pens 0-7. There are only 8 simultaneous colors on the EGA color model when using this mode. The separate alpha/graphics mode provides 16 simultaneous colors in the graphics plane.

Custom Symbol Sets and Fonts. HP BASIC normally uses the HP Roman-8 symbol set and 8×14 font. You can use the special file name "HPWFONT" to change the symbol set or font HP BASIC uses.

If you are using the combined alpha/graphics display mode you can specify the symbol set or font HP BASIC should use by creating a file named "HPWFONT" in the directory where HP BASIC was installed. HPWFONT should contain an 8×14 font as delivered with an EGA or similar adapter. The format of HPWFONT is the same as that of the standard MS-DOS EGA adapter products.

If you are using the separate alpha/graphics mode, the *existence* of HPWFONT in the directory where HP BASIC was installed will cause HP BASIC to use the standard MS-DOS symbol set and font provided by your display adapter. Note that HPWFONT does not have to contain anything, its existence is sufficient to change the font.

If you are using an EGA display in the separate alpha/graphics mode, the font used by HP BASIC will be the one you establish when you set up your EGA system. The file "HPWFONT" is ignored if it exists in the directory where HP BASIC was installed. The EGA system provides tools and a selection of font files which allow you to change the font.

To summarize, the video emulator supports these display adapters and modes:

PC Display Type	PC Resolution	Default Display Model	HP BASIC Resolution	Notes
Multimode	640 × 400	9836A	512 × 390	
Monochrome Plus	720 × 348	9816B	400 × 300	Can use 9836A with the loss of the top 42 dot-rows
EGA	640 × 350	9836C	512 × 350	Top 40 dot-rows are lost

The MS-DOS Communica- tions Port

The MS-DOS communications port (select code 19) allows the HP BASIC language processor programs to communicate with your computer's software. Because of this port, no new keywords need to be defined.

Communications Port Control Commands

The MS-DOS port commands BACKGROUND, EXIT, HPWSTATUS, SAVE_MODE_ON, SAVE_MODE_OFF, WAIT_ON, and WAIT_OFF are discussed in the following sections.

BACKGROUND. The BACKGROUND command allows you to leave an HPBASIC program running in background while you operate other software on your computer. HP BASIC programs running in background can access a printer at select code 26 and activate the PC beeper. However, any attempt to access any other PC resources will cause the program to be suspended until you return to HPBASIC. You can enter background by typing:

```
OUTPUT 19;"BACKGROUND"
```

The first time OUTPUT 19; "BACKGROUND" is executed, a memory resident driver is loaded. This driver requires approximately 8K bytes of memory.

EXIT. The EXIT command allows an orderly return from the HP BASIC language processor to your computer. You can return to the HP BASIC language processor at any time (before turning your computer off) by typing HPBASIC (Enter).

Caution 🤚

To protect file integrity, you must explicitly close all HP BASIC files before exiting HP BASIC. The recommended method of accomplishing this is to stop all HP BASIC programs before executing the EXIT command.

You can exit HPBASIC in two ways:

- Press EXIT*
- Type `OUTPUT 19; "EXIT"` (Enter)

HPWSTATUS. The HPWSTATUS command will provide useful information to help you set up the VIEWPORT parameters. You can include the following statements in a program:

```
OUTPUT 19;"HPWSTATUS"
ENTER 19;A$,B$,C,D$,E
```

The parameters in the ENTER statement are variables in your program. Upon completion of the above statements, the variables will contain the following information:

- `A$`—The version number of the HP BASIC Language Processor software you are using.
- `B$`—The viewable size of your display in the format nnnxnnn, (horizontal × vertical).
- `C`—The language processor card hardware version number.
- `D$`—This is taken from the MACHINE: keyword in your configuration file, for example C9836.
- `E`—The x/y (horizontal/vertical) ratio of the physical dot size on the display currently being used.

SAVE_MODE_ON. The HP BASIC alpha screen is saved and then restored on return to HP BASIC. However, information on the graphics screen is not saved. If you are executing an MS-DOS command that does *not* write to the display (such as `DIR > HPW_PIPE`), you may retain the graphics information by first executing `OUTPUT 19; "SAVE_MODE_OFF"`. You can return to the default state by executing `OUTPUT 19; "SAVE_MODE_ON"`.

---

* Refer to appendix E or the Key Function and Switch Configuration Guide for the specific keystrokes.

The following table may help to clarify what happens to the various screens with SAVE_MODE_ON and SAVE_MODE_OFF.

	HP BASIC Alpha Screen	HP BASIC Graphics Screen	MS-DOS Display Output
SAVE_MODE_ON	Retained	Cleared	Readable
SAVE_MODE_OFF	MS-DOS overwrites	Retained	Improperly formatted

As long as MS-DOS does not write to the screen, SAVE_MODE_OFF is the best to use; otherwise SAVE_MODE_ON is the best.

WAIT_ON. After you execute OUTPUT 19; "<MS-DOS Command>", the system will wait for a key to be pressed before returning to HP BASIC. If you do *not* want to wait for a key press, execute OUTPUT 19; "WAIT_OFF" first. You can restore the delay by executing OUTPUT 19; "WAIT_ON". The initial value of this parameter is established in the configuration file. The default configuration file establishes WAIT_ON as the initial value.

MS-DOS Commands

Any commands other than BACKGROUND, EXIT, HPWSTATUS, SAVE_MODE_ON, SAVE_MODE_OFF, WAIT_ON, or WAIT_OFF will be processed by the MS-DOS command shell, either PAM or COMMAND.COM.

For example, to execute a DIR command from the keyboard, type:

```
OUTPUT 19;"DIR"
```

The directory will be displayed on the screen. Press any key to restore the HP BASIC display. Note that the directory listed may not be the same as the current directory for HP BASIC.

`OUTPUT 19; "DIR"` will list the files on the drive that was most recently accessed by the HP BASIC emulator. To insure the list is from the drive you want, execute an `MSI` command for that drive immediately *before* you execute the `OUTPUT 19; "DIR"`.

`OUTPUT 19; "CD"` will *not* change the mass storage unit specifier (MSUS) that is active in HP BASIC.

If there are errors in starting the command, you will see an MS-DOS error message on the display.

The execution of MS-DOS applications through the MS-DOS communications port is not supported.

Using the ENTER Statement

ENTER can be used to obtain results of communications port control commands and MS-DOS commands. You can use EN-TER as shown in a program called ENTERDEMO on your manual examples disc. For example, ENTERDEMO has an OUTPUT 19; "DIR > HPW_PIPE" statement followed by an ENTER 19, B$ statement. If the MS-DOS file COMMAND.COM cannot be found, the ENTER 19;B$ statement will provide the following message:

```
ERROR: return value of -1 for DOS COMMAND "DIR > HPW_PIPE"
```

If an error occurs when the command is being executed, an MS-DOS error message will appear on the screen just as it does when you are running from MS-DOS.

If there are no errors, the next ENTER from select code 19 will get bytes from a file called HPW_PIPE. Subsequent EN-TERs continue to draw bytes from this file.

Executing an ENTER on select code 19 will produce a timeout if no file named HPW_PIPE exists, or if the file exists but is empty. You must create and fill an MS-DOS file named HPW_PIPE. This file is *not* created by HP BASIC.

Thus, a communications protocol is set up. MS-DOS programs can open and direct output to the HPW_PIPE file. The output can be read by the HP BASIC language processor.

The following program segment shows one way of using this capability within a program.

```
DIM Result$[80]
   .
   .
OUTPUT 19; "DIR > HPW_PIPE"
ENTER 19; Result$
   .
   .
```

The string variable Result$ will contain the first line of the output of the MS-DOS DIR command.

HP BASIC File System

A feature of the HP BASIC Language Processor is that it allows HP BASIC programs that work with the series 200/300 LIF (Logical Interchange Format) file system to use the MS-DOS file system. As a result, using the MS-DOS file system with the HP BASIC Language Processor requires an understanding of the series 200/300 file system, the MS-DOS file system, and the means by which the language processor system links the two.

The Series 200/300 LIF File System

The series 200/300 mass storage is based on external HP-IB disc drives; a series of such drives is connected to the workstation and accessed one at a time. Refer to figure B-1.
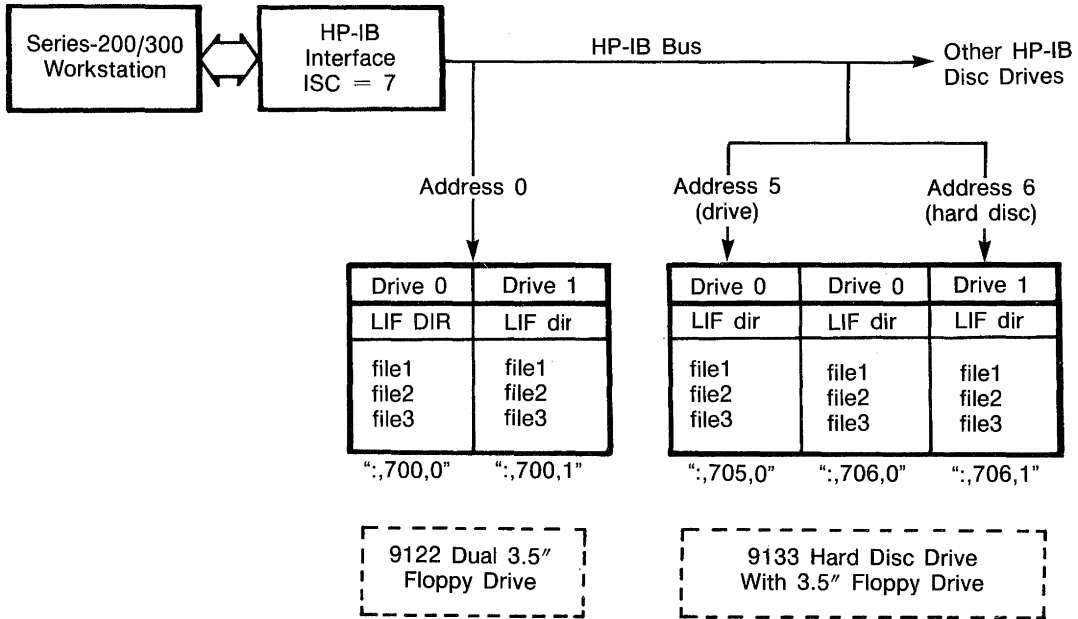
Figure B-1. Series 200/300 Mass Storage Scheme

The HP-IB drives can be flexible disc drives, hard disc drives, or combination flexible and hard disc drives. Each drive is designated by a separate address on the HP-IB bus. A combination drive may have two addresses; one for the flexible, another for the hard disc.

Each drive in the mass storage system contains its own file system in an HP-unique format called LIF (Logical Interchange Format). Each LIF drive contains a directory and a number of files.

HP BASIC selects each drive in the series 200/300 mass storage system through a Mass Storage Unit Specifier (MSUS, also sometimes called a Mass Storage Volume Specifier, or MSVS). An MSUS has the form:

":<drive type>, <drive select code>"

The default drive type applies to internal drives used with the HP BASIC Language Processor, so the <drive type> field can be left blank.

The <drive select code> is a three or four-digit number; the two least significant digits give the HP-IB address of the disc drive, and the one or two most significant digits give the interface select code of the series 200/300 HP-IB interface that the drive is connected to.

An HP BASIC user uses the MASS STORAGE IS statement to switch between different LIF drives; once a LIF drive has been selected, its files can be listed, copied, deleted, read, or written. MASS STORAGE IS has the syntax:

MASS STORAGE IS <MSUS>

In the example in figure B-1 (which illustrates a typical series 200/300 file system), a dual flexible disc drive (with HP-IB address 0) and a combination flexible and hard disc drive (with address 5 for the flexible disc drive and address 6 for the hard disc drive; note that the hard disc is further divided into two partitions) are connected to a series 200/300 HP-IB interface with interface select code 7.

The flexible disc drive on the combination drive can be selected with the statement:

```
MASS STORAGE IS ":,705,0"
```

The second partition of the hard disc drive could later be selected with:

```
MASS STORAGE IS ":,706,1"
```

Files on drives other than the one currently in effect can be accessed by appending the MSUS of the drive they are stored on to their filenames; for example, given the mass storage system shown in figure 1:

```
LOAD "SomeProg:,700,1"
```

would load a file from the flexible drive, even if it hadn't been selected by MASS STORAGE IS.

The MS-DOS File System

Most MS-DOS-based personal computers usually do not use external drives; they use built-in flexible disc and hard disc drives. A typical PC uses one or two flexible disc drives and a single hard disc drive. A hard disc drive may be partitioned into multiple volumes.

A: Drive	B: Drive	C: Drive	D: Drive	Z: Drive

```
┌─────────────┐  ┌─────────────┐  ┌─────────────┐  ┌──────────────────────────────┐
│ 360K Internal│  │ 1.2M Internal│  │   Internal  │  │ Other Hard Disc Volumes, Other│
│ Floppy Drive │  │ Floppy Drive │  │  Hard Disc  │  │ Hard Discs, Network Disc Servers,│
└─────────────┘  └─────────────┘  └─────────────┘  │ RAM Discs, HP-IB & HP-IL Disc │
                                                     │ Drives, ETC.                  │
                                                     └──────────────────────────────┘
```

C: Drive File System (Detail)

```
                    ┌──────────────┐
                    │ C:\          │
                    │ file1        │
                    │ file2        │
                    │ file3        │
                    └──────────────┘
        ┌───────────────┬───────────────────┐
        ▼               ▼                   ▼
┌──────────────┐ ┌──────────────┐        Other
│ C:\APPS      │ │ C:\BIN       │    Subdirectories
│ file1        │ │ file1        │
│ file2        │ │ file2        │
│ file3        │ │ file3        │
└──────────────┘ └──────────────┘
    ┌──────────────┐
    ▼              ▼
┌───────────────┐ ┌───────────────┐
│ C:\APPS\PGMS  │ │ C:\APPS\DATA  │
│ file1         │ │ file1         │
│ file2         │ │ file2         │
│ file3         │ │ file3         │
└───────────────┘ └───────────────┘
```
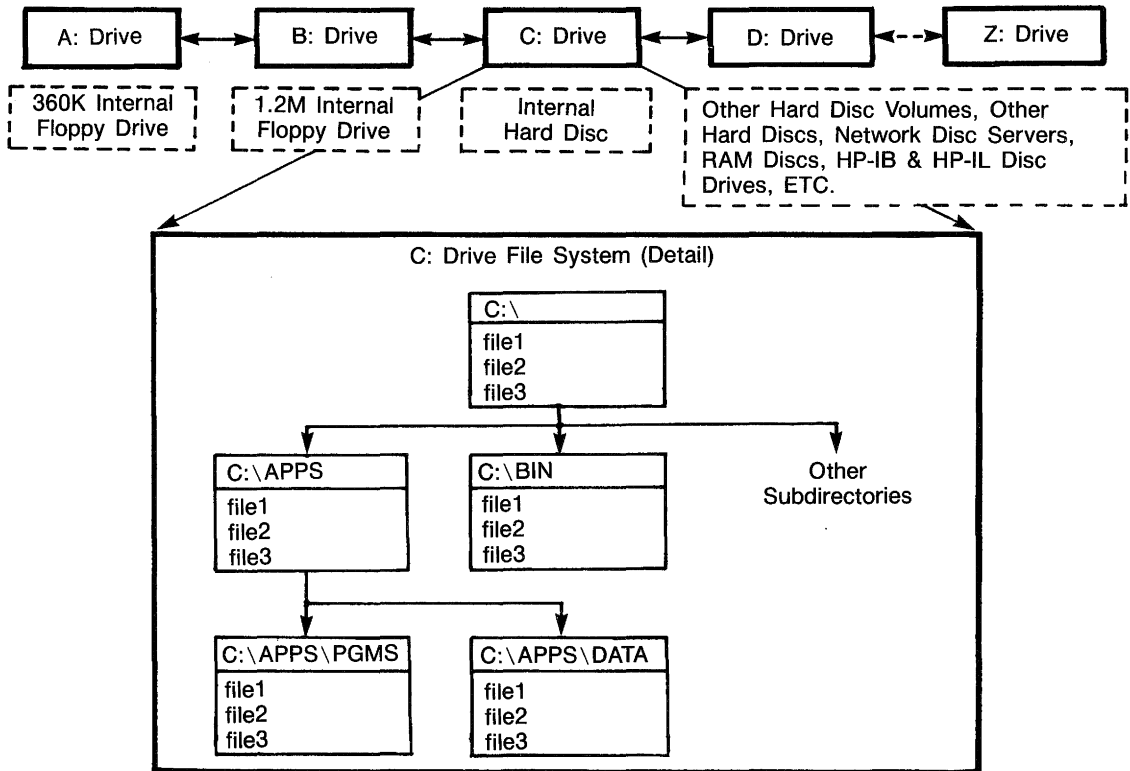
Figure B-2. MS-DOS Mass Storage Scheme

The MS-DOS file system can accommodate a large variety of mass storage systems:

- RAM discs.
- Network mass storage servers.
- External disc drives.

MS-DOS designates each of these drive with a drive ID code, consisting of a letter followed by a ":", giving drive IDs from "A:" to "Z:" (refer to figure A-2). In a typical system, the A and B drive IDs designate flexible drives, and the C drive ID specifies the hard disc. All you have to do to select a particular drive is enter its drive ID; for example, entering:

```
A:
```

selects a flexible disc drive. A file on a disc other than the one currently in effect can be identified by preceding its file name with the drive ID of the disc it's stored on; for example, entering the command:

```
COPY G:OLDFILE NEWFILE
```

copies a file named OLDFILE from drive G: to a file named NEWFILE in the disc drive and directory currently in effect.

Each MS-DOS disc contains its own hierarchical file system. A series 200/300 LIF volume is flat; it contains a single directory and a list of files. In a hierarchical file system, each disc contains a directory with the name "\", which may may contain files - as well as other sub-directories, which may contain files, and their own subdirectories...and so on, forming an upside-down tree of directories, with the "\" directory at the root of the tree. (As a result, the "\" directory is generally called the root directory, or simply root.)

The advantage of the hierarchical file system is that it makes it much easier for users to organize their files; files can be stored in subdirectories appropriate to their functions. For example (referring to figure B-2), two subdirectories could be created on a hard disc with drive ID "C:", one (named APPS) to store applications programs , the other (named BIN) to store various system programs. These subdirectories can be created with the MS-DOS MKDIR command:

```
MKDIR \BIN
MKDIR \APPS
```

Two further subdirectories could be made for the APPS directory, one to store auxiliary programs (PGMS) and another to store data files (DATA):

```
MKDIR \APPS\PGMS
MKDIR \APPS\DATA
```

Note that the two subdirectories are identified by linking their names to that of their parent directory, APPS, with a "\". Using this same convention, (and prefixing a drive ID) allows a file named "DFILE" in the "\APPS\DATA" directory to be designated by:

```
C:\APPS\DATA\DFILE
```

This is called the full pathname of the file (since it specifies a path through the directory tree), and it uniquely describes the file from any directory within the MS-DOS file system.

An MS-DOS user switches between directories with the CD (Change Directories) command. For example, if the current directory is "\APPS\PGMS", then:

```
CD \
```

changes the current directory to the root directory.

Language Processor Mass Storage Options

Since the language processor emulates computers that use a LIF file system but resides on a MS-DOS machine, understanding the file system can be difficult. Refer to figure B-3.

HP BASIC Configuration
Program (CONF)

:,1500,0 A:
:,1500,1 C:\BASIC
:,1500,2 C:\BASIC\PGMS
:,1500,3 C:\BASIC\DATA
:,1500,4 D:

BASIC
Language
Processor

HP-IB
Interface
ISC = 7,24,25

HP-IB Bus

To LIF Discs

Virtual HP-IB
Interface
ISC = 15

HPW.CON File

Language Processor Emulation Software (Running on PC Host)

LIF-MSUS-to-DOS-Directory Mapping Table

":,1500,0"	":,1500,1"	":,1500,2"	":,1500,3"	":,1500,4"
A:	C:\BASIC	C:\BASIC\PGMS	C:\BASIC\DATA	D:

A: Drive	B: Drive	C: Drive	D: Drive
Floppy Drive	Floppy Drive	Hard Disc	RAM Disc

C:\
DOS Files

C:\BASIC
HPWLIF.DIR
HPW Files
DOS Files

C:\BIN
DOS FILES

C:\BASIC\PGMS
HPWLIF.DIR
HPW Files
DOS Files

C:\BASIC\DATA
HPWLIF.DIR
HPW Files
DOS Files

Figure B-3. HP BASIC Access to MS-DOS File System

The language processor system can access mass storage in several ways:

- HP-IB disc drives can be connected to the language processor's built-in HP-IB interface (at interface select code 7) and accessed just as they are under series 200/300.
- Similarly, if the Language Processor has direct control over PC HP-IB interfaces (using interface select codes 24 or 25) these interfaces can be used to directly access HP-IB disc drives, just as they are by the built-in HP-IB interface.
- The Language Processor can access LIF flexible discs from built-in PC flexible drives.
- The Language Processor can also access files from MS-DOS directories using a scheme by which the MS-DOS directories appears to the language processor as virtual LIF drives.

In the first two cases, only LIF drives can be accessed by the language processor; this form of access is exactly the same as the series 200/300 scheme and won't be discussed further here. The last two cases, however, require further explanation, and will be discussed in the next section.

Concepts of the Language Processor Access to the MS-DOS File System. The language processor accesses MS-DOS drives and directories as drives of a virtual HP-IB disc drive (at address 0) connected to a virtual HP-IB interface (at interface select code 15.) Which MS-DOS drives and directories HP BASIC can access depends on how the language processor system is configured.

The configuration of the language processor system is controlled by a configuration file that specifies a list of MS-DOS directories that the language processor maps into HP BASIC virtual-LIF volumes. This configuration file is called HPW.CON and it can be modified using the configuration utility program CONF. This will be discussed later, or you can refer to appendix F for information on using this utility.

Each drive or directory is mapped into a virtual-LIF volume corresponding to its position in the list; the first item in the list becomes virtual-LIF volume 0, the second becomes virtual-LIF volume 1, and so on. The list can contain up to 15 drive or directory names, providing up to 15 virtual-LIF volumes numbered 0 through 14.

The default drive mapping (when no drives or directories are specified in the configuration file) is:

A: ":,1500,0"
B: ":,1500,1"
C: ":,1500,2"
D: ":,1500,3"

Assume that you want to change this mapping, and you want to map the following physical disc drives and directories into virtual-LIF volumes:

1. A:

2. C:\HP_BASIC

3. C:\HP_BASIC\PGMS

4. C:\HP_BASIC\DATA

5. D:

Execute the configuration utility program from MS-DOS and then move the cursor down the main menu to the number that appears to the right of the word "drives". Then press (F1), "EDIT DRIVES". The screen will display the 15 virtual-LIF volumes available, 0 through 14, on the left margin. These cannot be changed, however any mapping that exists for any of these volumes can be changed. To specify the desired mapping, position the cursor to the right of the appropriate LIF volume, enter the physical drive name, and press (Enter). The cursor will automatically move to the next line. If you do not wish to use the LIF volume on that line, simply press (Enter). To specify the mapping for the drives and directories listed above, the screen would look like this:

```
:,1500,0 A:
:,1500,1 C:\HP_BASIC
:,1500,2 C:\HP_BASIC\PGMS
:,1500,3 C:\HP_BASIC\DATA
:,1500,4 D:
:,1500,5
:,1500,6
:,1500,7
:,1500,8
:,1500,9
:,1500,10
:,1500,11
:,1500,12
:,1500,13
:,1500,14
```

When you are done, press (F8), "Return to MAIN". From the
main screen, press (F5) (SAVE CONFIG) to save the configu-
ration of the drives. Now boot HP BASIC and your
configuration is active.

To select drive C:\HP_BASIC\PGMS from HP BASIC, type:

```
MSI ":,1500,2"
```

and press (Enter). Under this scheme, LIF flexible discs can be
inserted into the PC's internal flexible disc drives and
accessed just as they would be in an HP-IB disc drive con-
nected directly to the language processor.

Access to MS-DOS discs and directories, however, is more complicated. To allow HP BASIC to access them in the same way that it accesses a LIF volume, a virtual LIF organization is installed on top of the MS-DOS disc or directory. The key to this virtual-LIF organization is a file named HPWLIF.DIR that exists on each MS-DOS disc or directory mapped into the HP BASIC file system. HPWLIF.DIR is an MS-DOS file whose data is organized in the same way as a conventional LIF directory; it contains a list of all the files HP BASIC has stored on that MS-DOS disc or directory. Furthermore, just as HPWLIF.DIR is a MS-DOS file with LIF organization, all the files HP BASIC stores on that disc or directory are MS-DOS files with LIF organization. These virtual-LIF files (and HPWLIF.DIR) are known as HPW files. All HPW files have internal organizations different from typical MS-DOS files; in certain cases they can be translated (using a utility called HPWUTIL, which is provided with the HP BASIC) so they can be accessed from MS-DOS programs.

Normal MS-DOS files can be stored on discs or directories that are being used as virtual-LIF volumes; there's no differ-ence between them and any other MS-DOS disc or directory, except that they contain certain unusual file types. Since these normal MS-DOS files are not listed in the virtual-LIF volume's HPWLIF.DIR file, HP BASIC cannot catalog or ac-cess them.

MS-DOS files cannot be read by HP BASIC even if they are recognized, since their internal organization differs from the organization of HP BASIC HPW files. (However, HPWUTIL can translate MS-DOS files into HPW files and add an appro-priate entry into HPWLIF.DIR.)

Implications of the Language Processor Access to MS-DOS File System. This unusual mass storage scheme (devised to allow two different file systems to coexist) has certain implications, some of which have already been mentioned (like the need to translate between MS-DOS and HPW formats). Other implications need to be explained in detail.

- When the HP BASIC language processor creates MS-DOS filenames, it uses these rules to map:
  - Embedded blanks and illegal MS-DOS characters become "_".
  - Lowercase letters are converted to uppercase letters.
  - Filenames greater than eight characters appear with MS-DOS extensions. For example, "ABCDEFGHIJ" would be renamed in MS-DOS as "ABCDEFGH.IJ".
  - To avoid duplicate names in MS-DOS, a unique two-letter extension is appended to the MS-DOS file name when needed.

- Attempting to access a LIF disc from MS-DOS will generate an error. Access LIF discs *only* from HP BASIC or HPWUTIL. Refer to appendix C for information on converting files using HPWUTIL.

- HP BASIC cannot recognize HPW files in a virtual-LIF volumes *unless* they have an entry in the volume's HPWLIF.DIR file. This means that an HP BASIC file cannot simply be copied into a virtual-LIF volume with the MS-DOS COPY command; HP BASIC won't recognize it, since COPY won't update HPWLIF.DIR. HP BASIC files should be copied from virtual-LIF volume to virtual-LIF volume by running HP BASIC and performing the copy through HP BASIC COPY command (the HPWUTIL program can be also be used for this purpose).

- Similarly, HPW files should not simply be deleted using the MS-DOS DEL command, since that does not remove their entry from the HPWLIF.DIR file. In fact, all HPW files are write-protected from MS-DOS to prevent this. However, in some cases, a user may want to clear the contents of a MS-DOS disc or directory designated as a virtual-LIF volume, and in that case the write-protection becomes a nuisance. Fortunately, write-protection can be removed using the MS-DOS ATTRIB command:

```
ATTRIB -R <filename>
```

You can use ATTRIB to remove write-protection from all files in the current MS-DOS directory as follows:

```
ATTRIB -R *.*
```

- Since virtual-LIF drives are simply MS-DOS directories containing HPW files, *any* directory in the MS-DOS file system - whether it's on a flexible disc, a hard disc, a PC RAM disc, a PC network mass storage server, an HP-IL disc drive, or an HP-IB disc drive - can be used as a virtual-LIF drive. (However, LIF discs can be accessed only from the PC's internal flexible drives.)
- If there is no HPWLIF.DIR file on the disc or directory designated as a virtual-LIF drive when HP BASIC attempts to access it, an HPWLIF.DIR file will automatically be created. Note that in this case an error will result if the disc in use is write-protected.

The HPWUTIL Utility Program. The HP HP BASIC language processor also includes a utility program called HPWUTIL, to allow the handling and conversion of HPW files in MS-DOS directories.

The HPWUTIL program allows you to:

- Initialize flexible discs in LIF format on the PC's flexible drives.
- Copy LIF discs to virtual-LIF volumes, and the reverse.
- Catalog LIF or virtual-LIF volumes.
- Check discs.
- Translate a MS-DOS file to an HPW file, or the reverse.
- Compress an HPWLIF.DIR directory file.

The HPWUTIL program can be run interactively or in a command-line mode; refer to appendix C for more information.

Using A PC HP-IB Mass Storage Device

If you have a separate HP-IB interface such as the HP 82990A, then you may use that interface as a second HP-IB interface from the HP BASIC Language Processor (the first is the built-in HP-IB interface on the language processor card). You can use this second HP-IB interface as an I/O channel using the emulation of select code 24 or 25. You may attach a mass storage device to this interface for use by the HP BASIC Language Processor. When accessed through select code 24 or 25, this mass storage device must operate using HP LIF-formatted discs.

You may also choose to connect a mass storage device to the PC HP-IB interface using MS-DOS formatted discs. In this case, use the configuration utility to allow HP BASIC to access the external device. Refer to appendix F.

Caution 🤚

When using an external PC HP-IB interface, access the interface using select code 15 if the interface is assigned to an MS-DOS drive. Access the interface using select code 24 if it is to be used by HP BASIC (either for instrument control or for LIF mass storage) and has no drive designator assigned by MS-DOS. You cannot have an external HP-IB interface assigned to an MS-DOS device *and* also use it as an instrument control interface. Be sure to access the interface using *either* select code 15 *or* select code 24, but never *both* 15 and 24.

Disc Initialization

Disc initialization in the HP BASIC language processor is not the same as in Series 200/300 HP BASIC. The HP BASIC language processor initializes floppy discs in MS-DOS format only (floppy discs can be initialized in LIF format using the disc utility HPWUTIL). To initialize a disc in the first drive specified in the DRIVES: keyword in the configuration file, type:

```
INITIALIZE ":,1500,0" (Enter)
```

HP BASIC will be temporarily suspended, and from this point on, the procedure will be the same as if you had entered the FORMAT command from MS-DOS. When the disc is initialized, HP BASIC will automatically return. A directory file is created on the initialized disc.

ID PROM

When using the 9836A and 9836C display models, the HP BASIC language processor emulates the standard Series 200 ID PROM. If your PC does not have an HP-HIL security module (HP 46084A) attached, then the serial number will be 11111111111, and not usable for security purposes. If your PC does have the security module attached, then the module will be used to generate a valid serial number for security purposes.

Using a Network With the HP BASIC Language Processor

A disc operating system which provides remote file access (RFA) in a transparent manner is often called a "networked disc operating system". When using such a system, the HP BASIC Language Processor can access remote file systems just as it would a local file system. If mass storage disc drive P is networked, then use the configuration utility to allow HP BASIC to access the networked drive. Refer to appendix F.

You can access networked PC printers using the select code 26 port to write to LPT1. If LPT1 is networked, then select code 26 will be also.

To access network services which provide network capabilities on a command basis (such as Network File Transfer (NFT) and Virtual Terminal (VT)), you must use the MS-DOS communication port (select code 19) from the HP BASIC Language Processor.

Access to the Shared Resource Management (SRM) network is provided using the SRM add-on interface card. Using this card and any set of MS-DOS networking services, the HP BASIC Language Processor can perform a gateway service between the SRM network and the provided MS-DOS network.

File Transfers

MS-DOS applications that need to read the HP BASIC Language Processor data files may do so directly since each such file is also recognized by MS-DOS. However, the internal format of the file must be known to the MS-DOS application reading the file.

MS-DOS applications that need to write to the HP BASIC Language Processor files must insure that such files are first copied out of the HPW file system using the utility program HPWUTIL. The files may then be modified by the MS-DOS application, and copied back into the HPW file system using HPWUTIL. Failure to follow this procedure may result in corruption of the file contents.

In general, format conversion is not performed during file transfer into or out of the HPW file system. File format conversion is usually the responsibility of the programmers. However, in the case of the HP BASIC Language Processor ASCII files, format conversion can be performed by HPWUTIL to and from MS-DOS .TXT files.

SOUND

The HP BASIC statement SOUND is not supported.

C

Utilities

The utilities program, HPWUTIL, provides utilities that simplify the conversion of files between Logical Interchange Format (LIF), MS-DOS format, and the file system used by HP BASIC. The program is menu-driven and operation is self-explanatory. It may be executed from a batch file or directly from MS-DOS. This appendix describes some of the conventions that are followed by the program and briefly summarizes each of the functions.

HP BASIC File Types

There are three file formats in the HP BASIC environment.

1. HP LIF format. Most floppy discs formatted on Series 200/300 machines running HP BASIC or HP PASCAL have this format.

2. HPW files. These are files created by HP BASIC on MS-DOS-formatted discs. They are MS-DOS files containing binary images of HP BASIC PROG, BDAT and other types of files. The MS-DOS file HPWLIF.DIR is a directory of all HPW files within the current MS-DOS directory. This file is automatically created by HP BASIC. HPW files and directories are write-protected from MS-DOS.

3. MS-DOS files. Any MS-DOS files that are not created by HP BASIC. These files cannot be accessed directly from within HP BASIC using standard keywords. Some MS-DOS files can be incorporated into the HP BASIC system by using this utilities program.

| Note | HPW and MS-DOS files may exist on the same disc. |

c

Utility Functions

The utility program is found on disc one. If you copied this disc onto a work disc or a hard disk, use that version. To execute the program from MS-DOS, type:

```
HPWUTIL (Enter)
```

A menu will appear on the screen allowing you to choose the utility function you want to perform. Each of these functions is described in the following sections.

| Note | You can execute certain functions of the utility program from a batch file. Complete information on this procedure can be found in the documentation file HPWUTIL.DOC on disc one. You can print this file or list it on your CRT using MS-DOS commands. |

(F1) Initializing an LIF Formatted Disc

This function initializes a floppy disc to the standard LIF format. For 5¼ inch discs, this function will work on the 1.2 Meg disc drive; however, for consistent results, we recommend that you use a 360K drive if one is available. Note that 3.5-inch discs are initialized to a capacity of 720K bytes, regardless of the normal capacity of the disc. After a disc is initialized by this function, it is compatible with floppy disc drives on Series 200/300 machines.

Note 👆 You must have BIOS ROM release A.01.05 or greater in order to use this function. This function will not work on computers with the older Phoenix BIOS ROM.

(F2) LIF Copy

This function copies an entire LIF disc to an MS-DOS directory and creates an HPW file subsystem and vice versa. The destination disc must have at least enough space to copy the files on the source medium. There are some differences in action depending on which way files are copied.

- LIF to HPW. When an LIF disc is copied to an HPW directory, the target directory is not overwritten. Instead, all files that are copied are appended to the end of the existing directory.
- HPW to LIF. In this case the entire destination disc is overwritten by the source files.
- HPW to HPW. In this case, the entire source directory is copied to the destination directory.

(F3) Catalog LIF or HPW Directory

This function performs a CAT on an LIF volume or HPW directory. The user must choose the format of the disc, either LIF or HPW. If the user specifies the wrong format, the read will fail. In this case, simply specify the other disc format and repeat the process.

(F4) Verify Disc/Directory Integrity

This function performs a check of the HPW file subsystem in a MS-DOS directory. It also shows MS-DOS file names of HP BASIC files. Note that the file names that you see when you execute CAT from HP BASIC may not be the same as the names of the MS-DOS files. Refer to appendix B for naming conventions.

This function also verifies that each file listed in the HPW directory is present as a non-empty file. If a discrepancy is found, you have the following options:

- Deleting the name of the missing file(s) from the HPW directory to make the system consistent. If you have a copy of the missing file, it may be restored to the HPW directory using the MS-DOS to HPW copy function (F5).
- Doing nothing, which leaves the subsystem inconsistent.
- In the case of a missing BDAT or ASCII HPW file, you can simply create an empty file.

The recommended action to take in case of a missing file is the first option above. This will avoid file access errors which could occur the next time you bring HP BASIC up.

(F5) Check In an MS-DOS File to an HPW Directory

This function allows you to incorporate an MS-DOS file into an HPW subsystem. You must specify the HP BASIC file type you want the HPW file to be. If the source file is an MS-DOS text file or an executable 68000 binary file, the program will do some transformation on the file. If the source file is an MS-DOS text file, it is converted into ASCII. The resulting file is an HPW file containing data in HP BASIC ASCII file format. If the source is a binary file that you intend to use to boot HP BASIC, you must supply the load and execution addresses. If the source file is a BDAT file, you must specify the logical record size in bytes.

The directory containing the source file and the destination directory containing the HPW file subsystem may or may not be the same. If the source and destination directories are not the same, a copy of the source file is created in the destination directory. If the directories are the same, a copy is *not* made (except when the source file is an MS-DOS text file).

If an MS-DOS text file is checked in as an HPW ASCII file, the format converted file is checked in and the source file remains.

Whether the destination file is copied or renamed, if a file with the same name exists in the destination directory, a unique name is generated by adding a suffix.

(F6) Check Out an HPW File to MS-DOS

This function checks out or exports a file from an HPW file subsystem and enables write access. Once a file is exported from an HPW file subsystem, it is not accessible from within HP BASIC unless it is checked in by the previous function.

No data conversion is done unless the file is ASCII type. In this case, the file can be converted into a MS-DOS text file. You can access the resulting file with standard MS-DOS functions like TYPE and MORE, and edit it with editors.

In order to make the original file type of the exported file easier to identify, a suffix is attached as the last character(s) of the MS-DOS file name extention.

These suffixes are:

A   ASCII type file when it is exported without format change.

TXT   ASCII type file when it is converted into a MS-DOS text file.

B   BIN file.

D   BDAT file.

P   Prog file

S   SYSTEM FILE.

X   Unknown file type.

Note

The previous two functions, (F5) and (F6), deal with file exchanges between MS-DOS and HPW file format and NOT between LIF and MS-DOS. In order to accomplish a LIF to MS-DOS exchange, you must first do a LIF conversion (F2).

(F7) Pack the HPW Directory File

This function deletes null entries left in the HPW directory file (HPWLIF.DIR) by the PURGE, SAVE, RE-STORE, and RE-SAVE commands, and then rebuilds the directory. The operation involves the directory file ONLY and regular files are not affected in any way, except that they may be renamed. As a result, subsequent mass storage operations in HP BASIC may run a little faster.

D List of Binaries

This appendix lists the binaries for the language extensions, CRT display drivers, mass storage device drivers, and the interface card drivers that are included with your system. All the binaries listed here are contained on disc three. This system will occupy 358K bytes of RAM. Remember that you can selectively load the binaries that you need to accomplish the task at hand, thus saving memory. Refer to chapter 2 for a description of how to load and save binaries in your system.

Language Extensions	Bytes RAM	Disc Space
CLOCK—Real time clock	4 K	4 K
COMPLEX—Complex number functions	7 K	7 K
EDIT—System editor*	20 K	20 K
ERR—Error messages	7 K	7 K
GRAPH—Basic graphics	46 K	43 K
GRAPHX—Graphics extensions	27 K	27 K
IO—Extended I/O	11 K	11 K
KBD—Keyboard extensions	13 K	12 K
KNB2_0—BASIC 2.0 knob default	0 K	0 K
LEX Lexical order	10 K	10 K
MS—Extended mass storage	9 K	9 K
MAT—Matrix operations	20 K	21 K

* These binaries are automatically loaded when you install HP BASIC. All other binaries may be selectively loaded, either during the install process or later.

Language Extensions	Bytes RAM	Disc Space
DCOMM—Data Communications	7 K	6 K
PDEV—Program development	13 K	13 K
SRM—Shared Resource Management	46 K	44 K
TRANS—Transfer statement	34 K	30 K
XREF—Cross reference statement	6 K	6 K

D

Drivers	Bytes RAM	Disc Space
CRTA—CRT display driver*	11K	4K
CRTB—CRT display driver	18K	6K
CRTX—CRT display driver	2K	2K
CS80—CS80 & SS80 disc driver*	11K	4K
DISC—AMIGO HP-IB disc driver*	7K	7K
GPIO—Parallel I/O driver*	6K	6K
HPIB—HP-IB built-in driver*	12K	12K
SERIAL—RS-232 driver	5K	5K

* These binaries are automatically loaded when you install
  HP BASIC. All other binaries may be selectively loaded, either
  during the install process or later.

E

Keyboard Information

This appendix describes the different keyboards that are supported by your HP BASIC system. Keep in mind that HP BASIC is emulating the ITF keyboard (such as the HP 46021), plus several functions from the HP 98203A keyboard and the HP 98203B & C keyboards.

These keyboards are shown in the following illustrations:
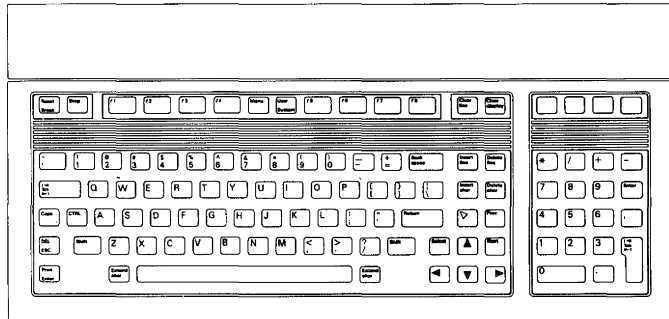
ITF Keyboard
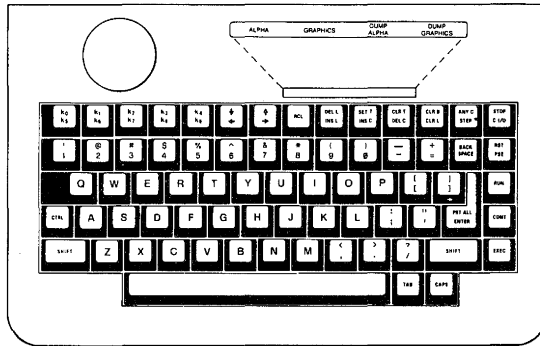
HP 98203A Keyboard
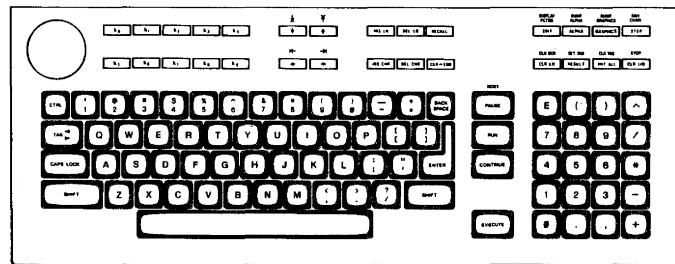
HP 98203B & C Keyboards

The following keyboards are supported by HP BASIC:

■ The Vectra PC keyboard.
■ The Enhanced Vectra PC keyboard.

These keyboards are shown in the following illustrations:
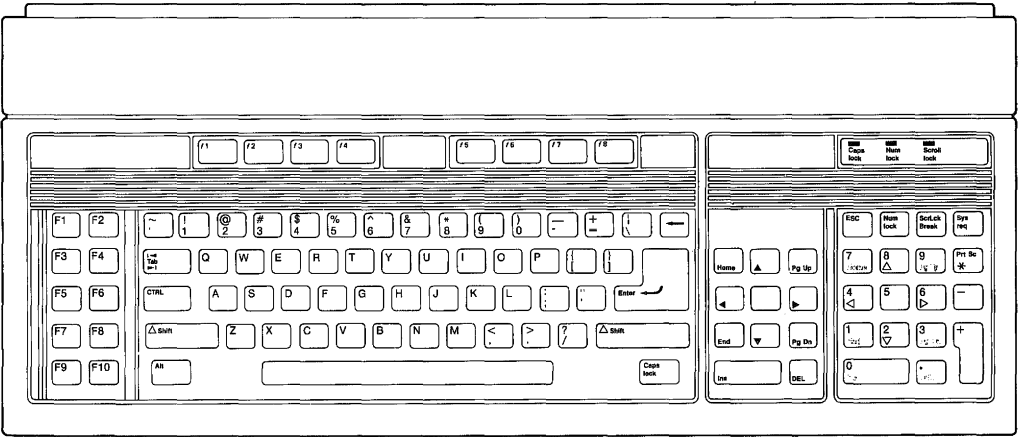
Vectra PC Keyboard
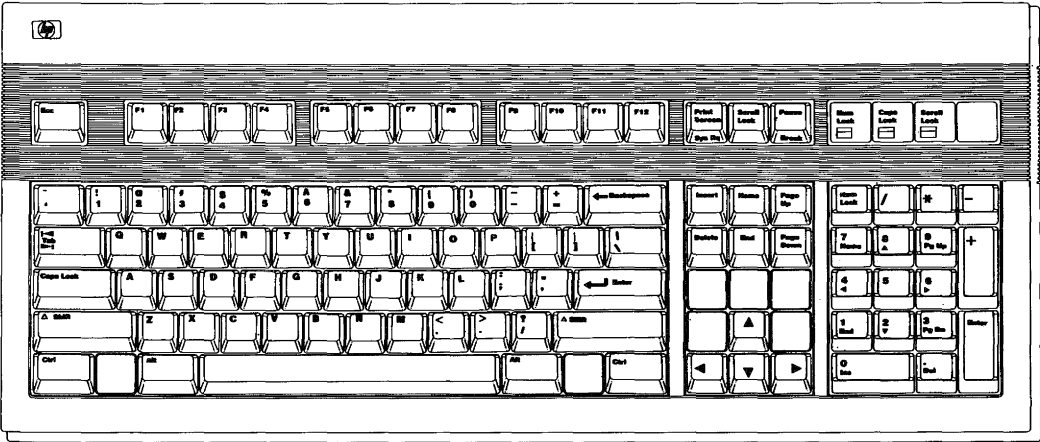
Enhanced Vectra PC Keyboard

From these illustrations determine the keyboard you are using, then go to the section that explains your particular keyboard.

Vectra PC Keyboard

The keys on the Vectra PC keyboard are arranged into the following functional groups:
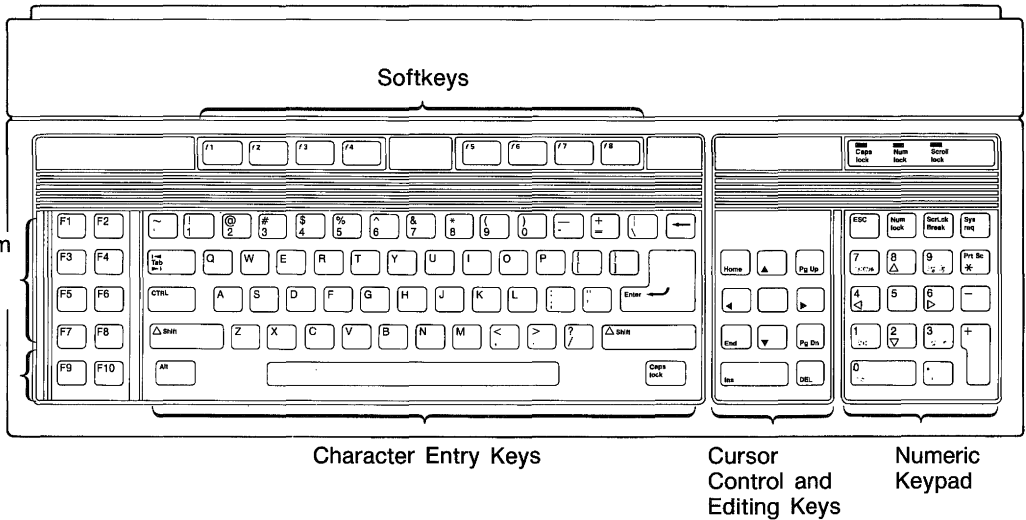
Vectra PC Keyboard Functional Groups

This section provides you with key definitions for the Vectra PC keyboard.

Note 👆 If you want to see the action of the keys demonstrated in the following steps, boot your HP BASIC system in the normal manner. Then type

SCRATCH (Enter)

before proceeding.

HP BASIC Keyboard Overlays

Two keyboard overlays designed for the Vectra PC keyboard are included with your HP BASIC system. Place the overlays on the keyboard as shown below:
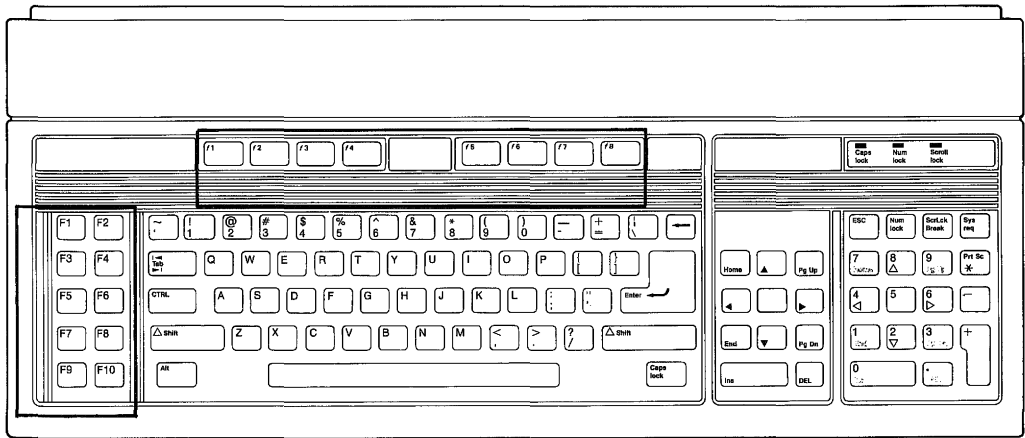
HP BASIC Keyboard Overlays

Character Entry
Keys
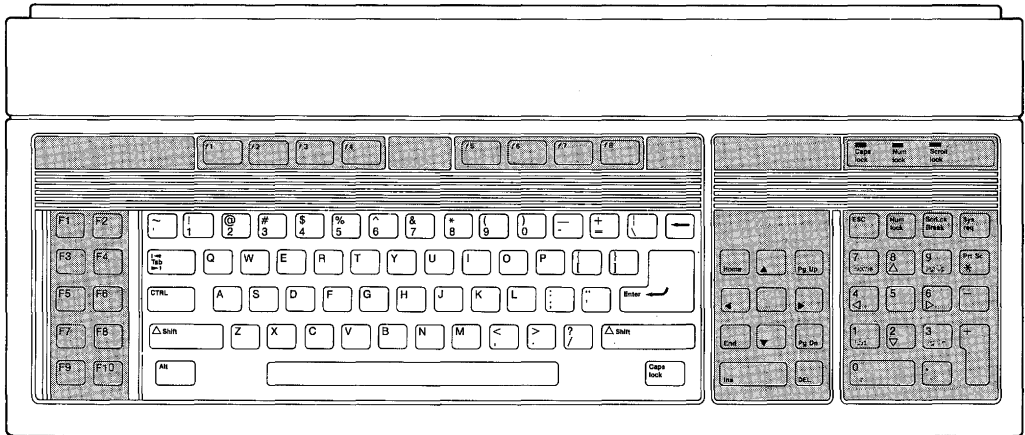
Vectra PC Keyboard Character Entry Keys

The character keys are arranged like a typewriter, but have some added features.

(Caps lock) This key sets the unshifted keyboard to either uppercase (the default after HP BASIC is booted) or lowercase. The computer displays which mode it is in when you press the (Caps lock) key.

Type in a few words, then press the (Caps lock) key and continue typing. Notice the case change. Press (CTRL)(End) when you are finished.

(Shift) You can use the (Shift) key to alternate between entering standard uppercase and lowercase letters. This is exactly the same as a typewriter.

(Enter)	The (Enter) key has three functions:
	■ When a program that is executing prompts you for data, respond by typing the requested data and then pressing (Enter). This signals the program that you have provided the data it requested and it can continue.
	■ When typing in lines of a program the (Enter) key is used to store each line of program code.
	■ After typing in a command the (Enter) key causes the command to be executed.
	Type EDIT and press (Enter). Notice the number 10 is displayed on the screen. This is the line number of the first line of an HP BASIC program. The computer is waiting for you to type in the line. Type:
	!FIRST LINE
	and press (Enter). The computer accepts the statement as a program line and displays 20 in preparation for the next line. Press (F1) when you are finished.
(Prt Sc)	Pressing (Shift)(Prt Sc) prints a copy of the alpha display on the default printer.
(Alt)	When pressed along with another key, this key allows you to generate the rest of the full 256-bit character set from the main typewriter section on Standard and European keyboards.
(Tab)	This key moves the cursor forward to preset tabs. Pressing (Shift)(Tab) moves the cursor backward to preset tabs.
	Before (Tab) can be used, a tab must be set. Tabs are set and cleared with system menu softkeys. This will be explained in the section entitled "System Softkeys."

(CTRL)   The control key works like the (Shift) key to access a set of standard control characters, such as line-feed and form-feed. These characters are useful to you for controlling some devices and communicating with other computers.

(Shift)(Enter)   The Select function ((Shift)(Enter)) beeps but performs no function unless it is program-defined.

Cursor Control Keys

Vectra PC Keyboard Cursor Control Keys

The cursor-control keys move the display cursor.

(▲)(▼)
(Shift)(▲)
(Shift)(▼)

The (▲) and (▼) keys allow you to scroll lines in the output area up and down. Shifted, the keys allow you to jump to the top and bottom of the output area.

(◄)(►)
(Shift)(◄)
(Shift)(►)

The (◄) and (►) keys allow you to move horizontally along a line. Shifted, they allow you to jump to the left and right limits of a line.

(←)

The backspace key works just like the (◄) key.

(Home)
(Shift)(Home)

The (Home) key positions the print position at the beginning position on the page. The shifted (Home) key places the print position at the beginning of the first empty line in the display (scrolls up if necessary). In edit mode, pressing this key (shifted or unshifted) causes the computer to beep. To verify operation of the (Home) key, press (Ctrl)(End). Then type PRINT "SOMETHING" and press (Enter); repeat twice. You should now have the following display:

SOMETHING
SOMETHING
SOMETHING

Press the (Home) key (unshifted).

Type PRINT "ANY " and press (Enter). Your display should look like this:

ANY THING
SOMETHING
SOMETHING

Press (CTRL)(End).

(Pg Up)
(Pg Dn)

In normal mode, pressing the (Pg Up) key causes the display to scroll down one page and pressing the (Pg Dn) key causes the display to scroll up one page. In edit mode, these keys move the display one-half page.

To test the horizontal movement of the cursor, type a few words and press the shifted and unshifted (◄) and (►) keys. Notice that the cursor cannot be moved beyond the characters you have typed. Press (Shift)(End) when finished.

To test the vertical movement of the cursor, type EDIT and press (Enter). Now type the following lines, pressing (Enter) after each line (the first line may be there already, so just press (Enter) to accept it):

```
10   !FIRST LINE
20   !SECOND LINE
30   !THIRD LINE
40   !FOURTH LINE
```

Try the shifted and unshifted (▲), (▼), and (Home) keys. Then try the (Pg Up) and (Pg Dn) keys. When you're done, press (F3) to exit. Then type SCRATCH and press (Enter) to clear memory.
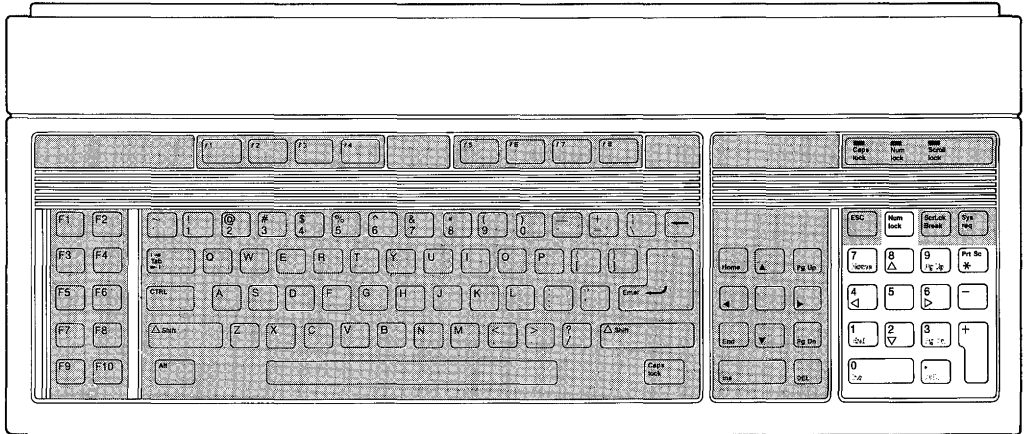
E

Numeric Keypad

The numeric keypad provides a convenient way to enter numbers and perform arithmetic operations. Simply type in the arithmetic expression you want to evaluate and press (Enter). The result is displayed in the lower-left corner of the screen.

Type in the following problem using the numeric keypad:

(26 + 14) / 4

Now press (Enter) to perform the calculation. The answer, 10, is displayed in the lower-left corner of the screen.

Editing Keys

E
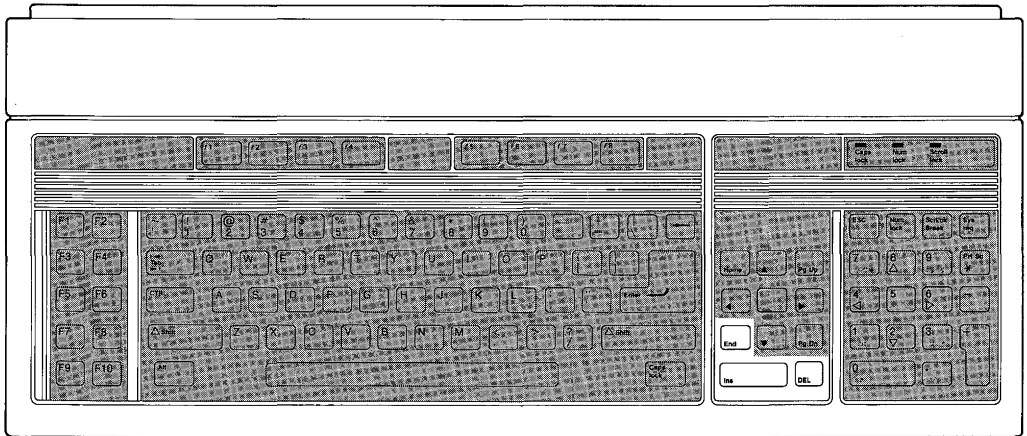
The editing keys put easy character editing and line editing at your
fingertips.

(Shift)(Ins)   Pressing (Shift) and (Ins) inserts a new line above
the cursor's current position (edit mode only).

Type EDIT, then press (Enter). Type in this line
(if it isn't already there):

10 !FIRST LINE

Now, with the cursor somewhere on line 10,
press (Shift)(Ins). Notice that a new line number
(1) is inserted before line 10. Press (F3) when
finished.

(Shift)(Del) Delete line	Pressing (Shift)(Del) deletes the line containing the cursor (edit mode only). Type EDIT, then press (Enter). Position the cursor to the line: 10 !FIRST LINE and press (Shift)(Del). The line is removed. To restore the line, press (F2) (or (Shift)(Pg Up)), then press (Enter) to enter it into the program. Press (F3) to exit edit mode.
(Ins) Insert char	Pressing (Ins) sets insert mode, allowing you to insert characters to the left of the cursor. Press the key a second time to cancel insert mode. Carefully type the following line exactly as shown: THIS IS A TEST Position the cursor under the period and press (Ins). Now type: OF INSERT MODE and press (Ins) again. The line should now look like this: THIS IS A TEST OF INSERT MODE. The new characters were inserted to the left of the period. Press (Shift)(End) when finished.
(Del) Delete char	Pressing (Del) deletes the character at the cursor's position. Type a few words and experiment with (Del), positioning the cursor at various places on the line. Notice that if you hold the key down, characters are deleted until you release it. Delete all of the characters you typed.

(End)	Pressing (End) clears from the current cursor position to the end of the line.
(Shift)(End)	Pressing (Shift)(End) clears the keyboard line and the message/results line.
	Type in a few words and use the (◄) key to position the cursor in the middle of the line. Press (End) to clear to the end of the line. Press (Shift)(End) to clear the rest of the line.
(CTRL)(End) Clr screen	Pressing (CTRL)(End) clears the entire alpha screen.

Type the following HP BASIC command:

`PRINT "PUT THIS MESSAGE IN THE OUTPUT AREA."`

Now press (Enter) to execute it. Press (F2) to recall the command, and press (Enter) again. Repeat this step several times to fill the screen with messages. Now press (CTRL)(End) to erase all lines at once.

E

Program Control Keys
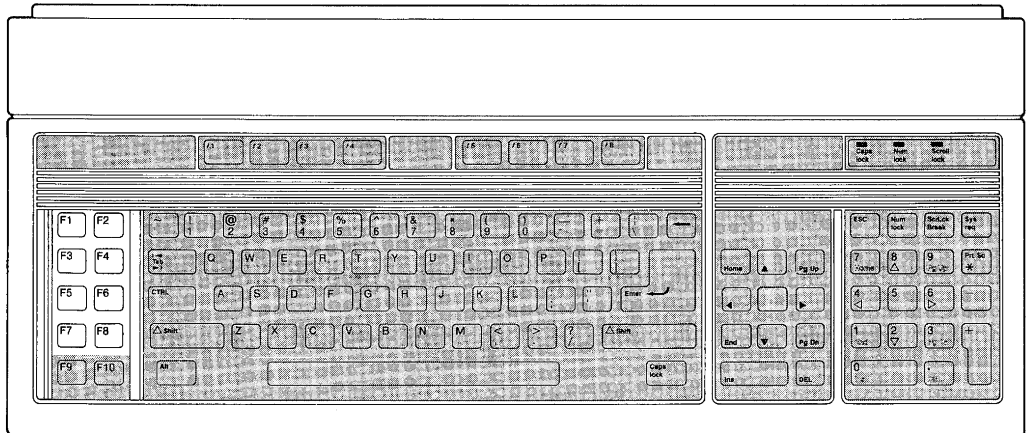
The following keys allow you to control execution of the program stored in the computer's memory.

(F1) Clr I/O	Pressing (F1) pauses program execution when the computer is performing or trying to perform an I/O operation. Press (F1) instead of (Shift)(F3) (Stop) when the computer is hung up on an I/O operation since (Shift)(F3) works only after the computer finishes the current program line. Pressing (F1) cancels the I/O operation and pauses the program at the current line.
(Shift)(F1) Reset	Pressing (Shift)(F1) pauses program execution immediately without erasing the program from memory. The HP BASIC Reset message indicates the computer is ready for your command.
(F2) or (Shift)(Pg Up) Recall	Pressing (F2) recalls the last line that you entered, executed, or deleted. Several previous lines can be recalled this way. Recall is particularly handy to use when you mistype a line. Instead of retyping the entire line, you can recall it, edit it using the editing keys, and enter or execute it again. Type: `PRINT "1" [Enter]` to print the number 1 on the screen. Now press (F2) to recall the print statement. Edit the statement to print the number 2 by positioning the cursor under the 1 and typing 2 over it. Press (Enter) again. Now press (F2) several times to see all of the statements it remembers. Then press (Ctrl)(End) to clear the screen when you are finished. (Shift)(F2) or (Shift)(Pg Dn) moves forward through the recall stack. Pressing (F8) in the System menu performs the same recall function as (F2).

(F3)	Pressing (F3) pauses program execution after the current line. Pressing Continue (F2) in the System menu resumes program execution from the point where it was paused.
Pause	

(Shift)(F3)
Stop

Pressing (Shift)(F3) stops program execution after the current line. To restart the program, press RUN ((f3)) in the System menu.

(F4)
RESULT

Pressing (F4) returns the result of the last arithmetic expression that was executed.

Press (Shift)(End) to clear the line, then type 23+45 and press (Enter).

The result, 68, is displayed in the lower-left corner of the screen. To add 123 to this value, press (F4) and type +123 amd press (Enter). The new result, 191, is now displayed. Press (Shift)(End) when finished.

(F5)
Step

(F5) allows you to execute one program line at a time. This is particularly useful for debugging programs.

(Shift)(F5)
Print All

The Print All key ((F5)) turns the printall mode on and off, allowing keyboard operations and displayed error messages to be copied to a printall device.

Press Print All once to set printall on and again to set printall off. The display's output area is the default printall device at powerup.

Press Print All to turn on printall mode. Now type in the following command:

```
PRINT "THIS IS A KEYBOARD
OPERATION"  (Enter)
```

Both the PRINT command and the message itself are displayed on the screen, which is the default printall device. Now type:

THIS WILL CAUSE AN ERROR (Enter)

Because this is not an executable HP BASIC statement, an error message is displayed at the bottom of the screen and in the printall area at the top of the screen. A log of all commands typed and executed at the keyboard, along with any error messages, is thus produced Press (Ctrl)(End) to clear the display, and press Print All ((F5)) to turn off printall mode.

(F7) Alpha	Pressing (F7) once turns on the alphanumeric display. Pressing it a second time turns off the graphics display. This key function requires that the GRAPH binary be loaded.
(Shift)(F7) Dump Alpha	Pressing (Shift)(F7) prints a complete copy of the alpha display on the default printer. The Dump Alpha function is also executed by (Shift)(Prt Sc).
(F8) Graphics	Pressing (F8) once turns on the graphics display. Pressing it the second time turns off the alphanumeric display.
(Shift)(F8) Dump Graph	Pressing (Shift)(F8) prints a complete copy of the graphics display on the default printer. Both Graphics and Dump Graph key functions require that the GRAPH language binary extension file be loaded.
(CTRL)(F9) Background	Pressing (CTRL)(F9) places HP BASIC in background operation. This is identical to executing OUTPUT 19;"BACKGROUND" from HP BASIC. Refer to appendix B for more information on background operation.
(CTRL)(F10) EXIT	Pressing (CTRL)(F10) terminates HP BASIC and returns your computer to MS-DOS.

Softkeys and
Softkey Control

There are eight softkeys (labeled (f1) through (f8)) and two keys ((F9) and (F10)) that control the definitions of the softkeys (MENU and SYSTEM).

When the HP BASIC system is booted, the softkeys default to System mode. The System mode menu appears at the bottom of your display. System softkeys are defined following control key definitions. In addition to the System mode, there are also three User modes: User 1, User 2, and User 3.

Softkey Control Keys. There are three control keys for the System, User, and Menu functions.

(F9)
Menu

Pressing (F9) toggles the softkey labels (turns them on if they're off and turns them off if they're on).

(Shift)(F9)

Pressing (Shift)(F9) increments user mode and menu *if* user mode is on.

(F10) System	Pressing (F10) causes softkeys to assume System mode. The System menu is displayed if the Menu key ((F9)) is toggled to the on position.
(Shift)(F10) User	Pressing (Shift)(F10) puts the softkeys in User 1 mode. The User 1 menu is displayed if the Menu key ((F9)) is toggled to the on position.

User menus are blank unless the KBD language extension binary is loaded.

Now let's get familiar with the control keys.

First we want to get the System mode selected and menu displayed. If the System menu is displayed, continue with the next paragraph. If it is not displayed, press (F10) (System). If it is still not displayed, press (F9) (Menu).

With the System menu displayed, press (F9) (Menu) several times. The system menu display should go on and off. Leave the System menu displayed and press (Shift)(F10) (User). The User 1 menu should appear on your display.

Press (Shift)(F9) (Shift Menu) several times. The displayed menus should rotate successively through the three User menus (User 1 → User 2 → User 3 → User 1 → User 2 →, etc.).

Press (F9) (menu) several times and the last User menu goes on and off. Leave the User menu on.

Finish this exercise by pressing (F10) (System) to get your softkeys back in System mode.

System Softkeys. The following paragraphs define the eight System softkeys.

(f1) Step	Step allows you to execute one program line at a time. This is particularly useful for debugging programs.
(f2) Continue	Continue resumes program execution from the point where it was paused by PAUSE ((F3)).
(f3) RUN	RUN starts a program running from the beginning.

(f4) Print All	The Print All key ((f4)) turns the printall mode on and off, allowing keyboard operations and displayed error messages to be copied to a printall device. Press (f4) once to turn printall on. Press it again to turn printall off. A message appears on the screen indicating whether printall is on or off.
(f5) and (Shift)(f5) Set Tab and Clr Tab	Set Tab ((f5)) sets a tab at the cursor's current position. Tabs remain in effect until cleared by either Clr Tab ((Shift)(f5)) or the SCRATCH A statement
	Clr Tab ((Shift)(f5)) clears a tab previously set at the cursor's position.
	Press the space bar to move the cursor forward a few spaces and press (f5) (Set Tab). Move the cursor back several spaces using (<), then pres (Tab). Move the cursor forward several more spaces with the space bar, then press (Shift)(Tab). To clear the tab, move the cursor to the unwanted tab position and press (Shift)(f5) (Clr Tab). Press (Shift)(End) when finished.
(f6) Display Functions	Pressing Display Functions ((f6)) turns on the display functions mode, allowing you to see special control characters (form-feed and carriage control, for example) on the screen. Pressing (f6) again turns the display functions mode off. An asterisk (*) appears to indicate that display functions is on.

Type the following line:

```
PRINT "DISPLAY FUNCTIONS
MODE OFF"
```

and press (Enter). Notice the display at the top of the screen. Now press Recall ((f8)) to recall the line, and edit it to read:

```
PRINT "DISPLAY FUNCTIONS ON"
```

Press Display Functions (⬚f6) and then press ⬚Enter. Notice that the carriage return and line feed control characters are now displayed. Press Display Functions (⬚f6) again to turn off the display functions mode. Press Clear Screen (⬚CTRL ⬚End) when you are finished.

⬚f7
Any char

Any char ((⬚f7)) is used to find any ASCII character. First press ⬚f7 (Any char). The following message appears above the menu:

```
Enter 3 digits, 000 to 255
```

Enter a three-digit number from 000 through 255 representing the decimal equivalent of an ASCII character. The computer automatically displays the character on the screen. For a list of characters and their equivalent decimal values, see the US ASCII Character Codes table in the "Useful Tables" appendix of the *BASIC Language Reference M–Z* (Volume 2).

Press ⬚f7 (Any char), then type 65 which is the decimal equivalent of "A". The display line now displays "A". Press ⬚Shift ⬚End to erase it.

⬚f8 or ⬚F2
or
⬚Shift ⬚Pg Up
Recall

The Recall softkey ((⬚f8) or ⬚F2 or ⬚Shift ⬚Pg Up)) acts just like system control key ⬚F10 described earlier in this section. Recall recalls the last line that you entered, executed, or deleted. Several previous lines can be recalled this way. Recall is particularly handy to use when you mistype a line. Instead of retyping the entire line, you can recall it, edit it using the editing keys, and enter or execute it again.

Type:

```
PRINT "1"
```

and press ⬚Enter to print the number 1 on the screen. Now press Recall ((⬚f8)) to recall the PRINT statement. Edit the statement to print the number 2 by positioning the cursor under the number 1 and entering the number 2. over it. Press ⬚Enter again. Now press Recall ((⬚f8)) to see all of the statements it remembers.

Note that Recall goes backward through the queue. Pressing (Shift)(f8) (or (Shift)(F2) or (Shift)(Pg Dn)) allows you to cycle forward through the queue until the last line entered, executed, or deleted is displayed.

Enhanced Vectra PC Keyboard

The keys on the Enhanced Vectra PC keyboard are arranged into the following functional groups:
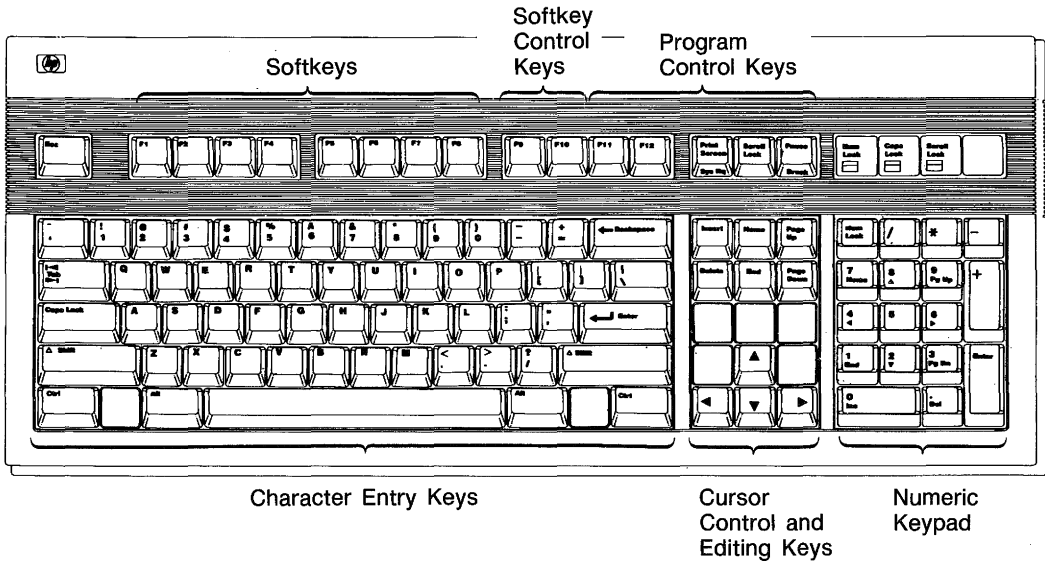
Enhanced Vectra PC Keyboard Functional Groups

This section provides you with key definitions for the Enhanced Vectra PC keyboard.

Note 👆 If you want to see the action of the keys demonstrated in the following steps, boot your HP BASIC system in the normal manner. Then type:

SCRATCH (Enter)

before proceeding.

HP BASIC
Keyboard Overlays

Two keyboard overlays designed for the Enhanced Vectra PC keyboard are included with your HP BASIC system. Place the overlays on the keyboard as shown below:
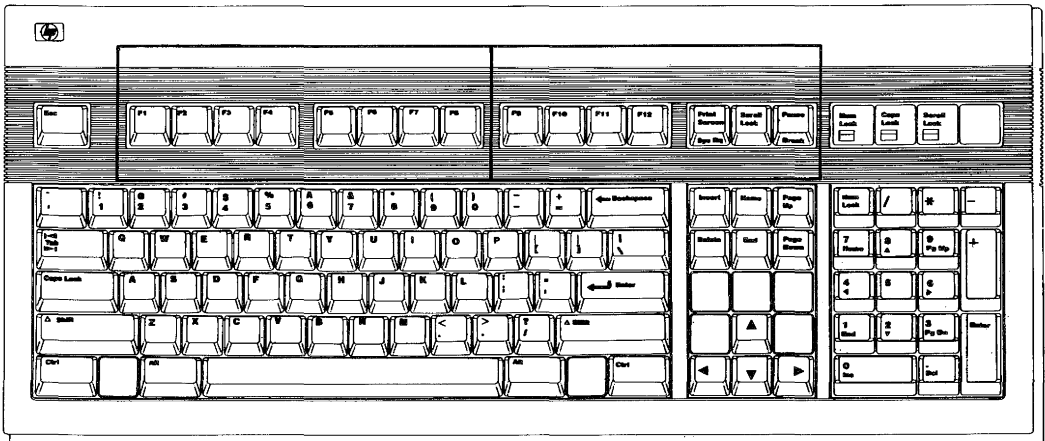
HP BASIC Keyboard Overlays

Character Entry Keys
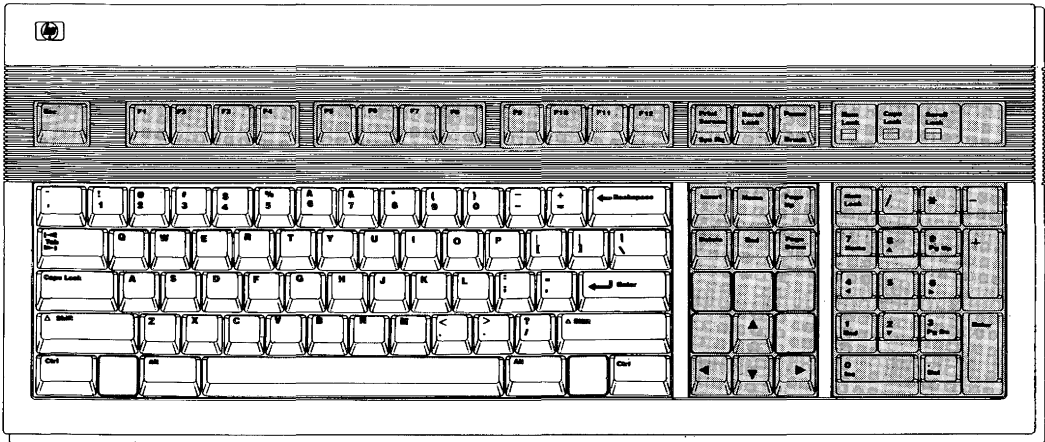
Enhanced Vectra PC Keyboard Character Entry Keys

The character keys are arranged like a typewriter, but have some added features.

(Caps lock)   This key sets the unshifted keyboard to either uppercase (the default after HP BASIC is booted) or lowercase. The computer does not display which mode the computer is in when you press the (Caps lock) key.

Type in a few words, then press the (Caps lock) key and continue typing. Notice the case change. Press (CTRL)(End) when you are finished.

(Shift)   You can use the (Shift) key to alternate between entering standard uppercase and lowercase letters. This is exactly the same as a typewriter.

(Enter)

The (Enter) key has three functions:

- When a program that is executing prompts you for data, respond by typing the requested data and then pressing (Enter). This signals the program that you have provided the data it requested and it can continue.
- When typing in lines of a program the (Enter) key is used to store each line of program code.
- After typing in a command the (Enter) key causes the command to be executed.

Type EDIT and press (Enter). Notice the number 10 is displayed on the screen. This is the line number of the first line of an HP BASIC program. The computer is waiting for you to type in the line. Type:

!FIRST LINE

and press (Enter). The computer accepts the statement as a program line and displays 20 in preparation for the next line. Press (F1) when you are finished.

(Alt)

When pressed along with another key, this key allows you to generate the rest of the full 256-bit character set from the main typewriter section on Standard and European keyboards.

(Tab)

This key moves the cursor forward to preset tabs. Pressing (Shift)(Tab) moves the cursor backward to preset tabs.

Before (Tab) can be used, a tab must be set. Tabs are set and cleared with system menu softkeys. This will be explained in the section entitled "System Softkeys."

(CTRL) The control key works like the (Shift) key to access a set of standard control characters, such as line-feed and form-feed. These characters are useful to you for controlling some devices and communicating with other computers.

(Shift)(Enter) The Select function ((Shift)(Enter)) beeps but performs no function unless it is program-defined.

Cursor Control Keys

Enhanced Vectra PC Keyboard Cursor Control Keys

The cursor-control keys move the display cursor.

Keys	Description
(▲)(▼) (Shift)(▲) (Shift)(▼)	The (▲) and (▼) keys allow you to scroll lines in the output area up and down. Shifted, the keys allow you to jump to the top and bottom of the output area.
(◄) (►) (Shift)(◄) (Shift)(►)	The (◄) and (►) keys allow you to move horizontally along a line. Shifted, they allow you to jump to the left and right limits of a line.
(←)	The backspace key works just like the (◄) key.
(Home) (Shift)(Home)	The (Home) key positions the print position at the beginning position on the page. The shifted (Home) key places the print position at the beginning of the first empty line in the display (scrolls up if necessary). In edit mode, pressing this key (shifted or unshifted) causes the computer to beep. To verify operation of the (Home) key, press (Ctrl)(End). Then type PRINT "SOMETHING" and press (Enter); repeat twice. You should now have the following display:

```
SOMETHING
SOMETHING
SOMETHING
```

Press the (Home) key (unshifted).

Type PRINT "ANY " and press (Enter). Your display should look like this:

```
ANY THING
SOMETHING
SOMETHING
```

Press (CTRL)(End).

(Page Up) In normal mode, pressing the (Page Up) key
(Page Down) causes the display to scroll down one page and
pressing the (Page Down) key causes the display
to scroll up one page. In edit mode, these keys
move the display one-half page.

To test the horizontal movement of the cursor,
type a few words and press the shifted and
unshifted (◄) and (►) keys. Notice that the
cursor cannot be moved beyond the characters
you have typed. Press (Shift)(End) when
finished.

To test the vertical movement of the cursor,
type EDIT and press (Enter). Now type the
following lines, pressing (Enter) after each line
(the first line may be there already, so just
press (Enter) to accept it):

```
10   !FIRST LINE
20   !SECOND LINE
30   !THIRD LINE
40   !FOURTH LINE
```

Try the shifted and unshifted (▲), (▼), and
(Home) keys. Then try the (Page Up) and
(Page Down) keys. When you're done, press
(Pause) to exit. Then type SCRATCH and press
(Enter) to clear memory.
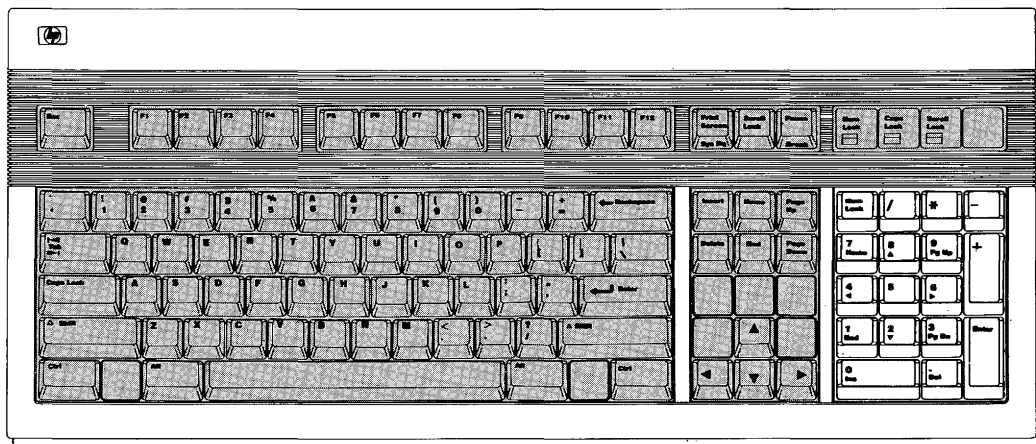
E

Numeric Keypad

The numeric keypad provides a convenient way to enter numbers and perform arithmetic operations. Simply type in the arithmetic expression you want to evaluate and press (Enter).
The result is displayed in the lower-left corner of the screen.

Type in the following problem using the numeric keypad:

(26+14)/4

Now press (Enter) to perform the calculation. The answer, 10, is displayed in the lower-left corner of the screen.

Editing Keys

E
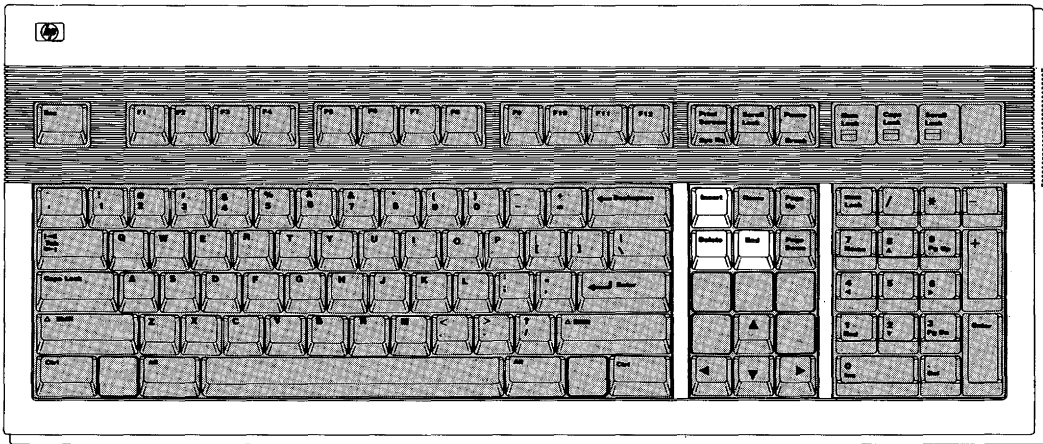
The editing keys put easy character editing and line editing at your
fingertips.

(Shift)(Insert)   Pressing (Shift) and (Insert) inserts a new line
above the cursor's current position (edit mode
only).

Type EDIT, then press (Enter). Type in this
line (if it isn't already there):

10 !FIRST LINE

Now, with the cursor somewhere on line 10,
press (Shift)(Insert). Notice that a new line
number (1) is inserted before line 10. Press
(Pause) when finished.

(Shift)(Delete) Delete line	Pressing (Shift)(Delete) deletes the line containing the cursor (edit mode only).

Type EDIT, then press (Enter). Position the cursor to the line:

10 !FIRST LINE

and press (Shift)(Delete). The line is removed. To restore the line, press (Shift)(Page Up), then press (Enter) to enter it into the program. Press (Pause) to exit edit mode.

(Insert) Insert char	Pressing (Insert) sets insert mode, allowing you to insert characters to the left of the cursor. Press the key a second time to cancel insert mode.

Carefully type the following line exactly as shown:

THIS IS A TEST .

Position the cursor under the period and press (Insert). Now type:

OF INSERT MODE

and press (Insert) again. The line should now look like this:

THIS IS A TEST OF INSERT MODE.

The new characters were inserted to the left of the period. Press (Shift)(End) when finished.

(Delete) Delete char	Pressing (Delete) deletes the character at the cursor's position.

Type a few words and experiment with (Delete), positioning the cursor at various places on the line. Notice that if you hold the key down, characters are deleted until you release it. Delete all of the characters you typed.

(End)	Pressing (End) clears from the current cursor position to the end of the line.
(Shift)(End)	

Pressing (Shift)(End) clears the keyboard line and the message/results line.

Type in a few words and use the (◄) key to position the cursor in the middle of the line. Press (End) to clear to the end of the line. Press (Shift)(End) to clear the rest of the line.

(Ctrl)(End)	Pressing (Ctrl)(End) clears the entire alpha screen.
Clr screen	

Type the following HP BASIC command:

```
PRINT "PUT THIS MESSAGE IN THE
OUTPUT AREA."
```

Now press (Enter) to execute it. Press (Shift)(Page Up) to recall the command, and press (Enter) again. Repeat this step several times to fill the screen with messages. Now press (Ctrl)(End) to erase all lines at once.

Program Control Keys

The following keys allow you to control execution of the program stored in the computer's memory.

(Scroll Lock) Clr I/O	Pressing (Scroll Lock) pauses program execution when the computer is performing or trying to perform an I/O operation. Press (Scroll Lock) instead of (Shift)(Pause) (Stop) when the computer is hung up on an I/O operation since (Shift)(Pause) works only after the computer finishes the current program line. Pressing (Scroll Lock) cancels the I/O operation and pauses the program at the current line.
(Shift)(Scroll Lock) Reset	Pressing (Shift)(Scroll Lock) (Reset) pauses program execution immediately without erasing the program from memory. The HP BASIC Reset message indicates the computer is ready for your command.
(Shift)(Page Up) Recall	Pressing (Shift)(Page Up) recalls the last line that you entered, executed, or deleted. Several previous lines can be recalled this way. Recall is particularly handy to use when you mistype a line. Instead of retyping the entire line, you can recall it, edit it using the editing keys, and enter or execute it again. Type:

PRINT "1" [Enter]

to print the number 1 on the screen. Now press (Shift)(Page Up) to recall the print statement. Edit the statement to print the number 2 by positioning the cursor under the 1 and typing 2 over it. Press (Enter) again. Now press (Shift)(Page Up) several times to see all of the statements it remembers. Then press (Ctrl)(End) to clear the screen when you are finished.

(Shift)(Page Down) moves forward through the recall stack.

Pressing (f8) in the System menu performs the same recall function as (Shift)(Page Up).

(Pause)

Pressing (Pause) pauses program execution after the current line. Pressing Continue ((f2)) in the System menu resumes program execution from the point where it was paused.

(Shift)(Pause)
Stop

Pressing (Shift)(Pause) stops program execution after the current line. To restart the program, press RUN ((f3)) in the System menu.

(Shift)(Print Screen)
Step

(Shift)(Print Screen) allows you to execute one program line at a time. This is particularly useful for debugging programs.

(Print Screen)
Print All

Print All ((Print Screen)) turns the printall mode on and off, allowing keyboard operations and displayed error messages to be copied to a printall device.

Press Print All (Print Screen) once to set printall on and again to set printall off. The display's output area is the default printall device at powerup.

Press (Print Screen) (Print All) to turn on printall mode. Now type in the following command:

```
PRINT "THIS IS A KEYBOARD
OPERATION"  (Enter)
```

Both the PRINT command and the message itself are displayed on the screen, which is the default printall device. Now type:

```
THIS WILL CAUSE AN ERROR  (Enter)
```

Because this is not an executable HP BASIC statement, an error message is displayed at the bottom of the screen and in the printall area at the top of the screen. A log of all commands typed and executed at the keyboard, along with any error messages, is thus produced. Press (Ctrl)(End) to clear the display, and press (Print Screen) (Print All) to turn off printall mode.

(CTRL)(F9) Background	Pressing (CTRL)(F9) places HP BASIC in background operation. This is identical to executing OUTPUT 19;"BACKGROUND" from HP BASIC. Refer to appendix B for more information on background operation.
(CTRL)(F10) EXIT	Pressing (CTRL)(F10) terminates HP BASIC and returns your computer to MS-DOS.
(F11) Alpha	Pressing (F11) once turns on the alphanumeric display. Pressing it a second time turns off the graphics display. This key function requires that the GRAPH binary be loaded.
(Shift)(F11) Dump Alpha	Pressing (Shift)(F11) prints a complete copy of the alpha display on the default printer.
(F12) Graphics	Pressing (F12) once turns on the graphics display. Pressing it the second time turns off the alphanumeric display.
(Shift)(F12) Dump Graph	Pressing (Shift)(F12) prints a complete copy of the graphics display on the default printer.
	Both Graphics and Dump Graph key functions require that the GRAPH binary extension file be loaded.

Softkeys and
Softkey Control

There are eight softkeys (labeled (F1) through (F8)) and two keys ((F9) and (F10)) that control the definitions of the softkeys (MENU and SYSTEM).

When the HP BASIC system is booted, the softkeys default to System mode. The System mode menu appears at the bottom of your display. System softkeys are defined following control key definitions. In addition to the System mode, there are also three User modes: User 1, User 2, and User 3.

Softkey Control Keys. There are three control keys for the System, User, and Menu functions.

(F9) Menu	Pressing (F9) toggles the softkey labels (turns them on if they're off and turns them off if they're on).
(Shift)(F9)	Presing (Shift)(F9) increments user mode and menu *if* user mode is on.
(F10) System	Pressing (F10) causes softkeys to assume System mode. The System menu is displayed if the Menu key ((F9)) is toggled to the on position.
(Shift)(F10) User	Pressing (Shift)(F10) puts the softkeys in User 1 mode. The User 1 menu is displayed if the Menu key ((F9)) is toggled to the on position.

User menus are blank unless the KBD language extension binary is loaded.

Now let's get familiar with the control keys.

First we want to get the System mode selected and menu displayed. If the System menu is displayed, continue with the next paragraph. If it is not displayed, press (F10) (System). If it is still not displayed, press (F9) (Menu).

With the System menu displayed, press (F9) (Menu) several times. The system menu display should go on and off. Leave the System menu displayed and press (Shift)(F10) (User). The User 1 menu should appear on your display.

Press (Shift)(F9) (Shift Menu) several times. The displayed menus should rotate successively through the three User menus (User 1 → User 2 → User 3 → User 1 → User 2 →, etc.).

Press (F9) (menu) several times and the last User menu goes on and off. Leave the User menu on.

Finish this exercise by pressing (F10) (System) to get your softkeys back in System mode.

System Softkeys. The following paragraphs define the eight System softkeys.

(F1)
Step

Step allows you to execute one program line at a time. This is particularly useful for debugging programs.

(F2)
Continue

Continue resumes program execution from the point where it was paused by (Pause).

(F3)
Run

Run starts a program running from the beginning.

(F4)
Print All

The Print All key ((F4)) turns the printall mode on and off, allowing keyboard operations and displayed error messages to be copied to a printall device. Press (F4) once to turn printall on. Press it again to turn printall off. A message appears on the screen indicating whether printall is on or off.

(F5) and
(Shift)(F5)
Set Tab and
Clr Tab

Set Tab ((F5)) sets a tab at the cursor's current position. Tabs remain in effect until cleared by either Clr Tab ((Shift)(F5)) or the SCRATCH A statement

Clr Tab ((Shift)(F5)) clears a tab previously set at the cursor's position.

Press the space bar to move the cursor forward a few spaces and press (F5) (Set Tab). Move the cursor back several spaces using (◄), then pres (Tab). Move the cursor forward several more spaces with the space bar, then press (Shift)(Tab). To clear the tab, move the cursor to the unwanted tab position and press (Shift)(F5) (Clr Tab). Press (Shift)(End) when finished.

(F6)
Display
Functions

Pressing Display Functions ((F6)) turns on the display functions mode, allowing you to see special control characters (form-feed and carriage control, for example) on the screen. Pressing (F6) again turns the display functions mode off. An asterisk (*) appears to indicate that display functions is on.

E

Type the following line:

```
PRINT "DISPLAY FUNCTIONS
MODE OFF"
```

and press (Enter). Notice the display at the top of the screen. Now press Recall ((F8)) to recall the line, and edit it to read:

```
PRINT "DISPLAY FUNCTIONS ON"
```

Press Display Functions ((F6)) and then press (Enter). Notice that the carriage return and line feed control characters are now displayed. Press Display Functions ((F6)) again to turn off the display functions mode. Press Clear Screen ((CTRL)(End)) when you are finished.

(F7)
Any char

Any char ((F7)) is used to find any ASCII character. First press (F7) (Any char). The following message appears above the menu:

```
Enter 3 digits, 000 to 255
```

Enter a three-digit number from 000 through 255 representing the decimal equivalent of an ASCII character. The computer automatically displays the character on the screen. For a list of characters and their equivalent decimal values, see the US ASCII Character Codes table in the "Useful Tables" appendix of the *BASIC Language Reference M–Z* (Volume 2).

Press (F7) (Any char), then type 65 which is the decimal equivalent of "A". The display line now displays "A". Press (Shift)(End) to erase it.

(F8) or
(Shift)(Page Up)
Recall

The Recall softkey ((F8) or (Shift)(Page Up))
acts just like system control key (F10) de-
scribed earlier in this section. Recall recalls
the last line that you entered, executed, or
deleted. Several previous lines can be re-
called this way. Recall is particularly handy
to use when you mistype a line. Instead of
retyping the entire line, you can recall it, edit
it using the editing keys, and enter or execute
it again.

Type:

```
PRINT "1"
```

and press (Enter) to print the number 1 on the
screen. Now press Recall ((F8)) to recall the
PRINT statement. Edit the statement to print
the number 2 by positioning the cursor un-
der the number 1 and entering the number 2.
over it. Press (Enter) again. Now press Recall
((F8)) to see all of the statements it remem-
bers. Note that Recall goes backward
through the queue. Pressing (Shift)(F8) (or
(Shift)(Page Down)) allows you to cycle forward
through the queue until the last line entered,
executed, or deleted is displayed.

Keyboard Mapping

The following table shows the relationships between the series 200/300 keyboards and the Vectra PC and Enhanced Vectra PC keyboards.

Function	HP 98203A,B,C Series 200	HP 46021A ITF Series 300	Vectra PC Keyboard	Enhanced Vectra PC Keyboard
ALPHA	(Alpha)	Unlableled Key 2	(F7)	(F11)
ANY CHARACTER	(Any Char)	(System) (f7)	(System) (f7)	(System) (F7)
BACKGROUND	*	*	(CTRL)(F9)*	(Ctrl)(F9)*
CLEAR TO END	(Clr▶End)	(Clr Line)	(End)	(End)
CLEAR I/O	(Clr I/O)	(Break)	(F1)	(Scroll Lock)
CLEAR LINE	(Clr Ln)	(Shift)(Clr Line)	(Shift End)	(Shift End)
CLEAR SCREEN	(Clr Scr)	(Clr Disp)	(CTRL End)	(Ctrl End)
CLEAR TAB	(Clr Tab)	(System) (Shift)(f5)	(System) (Shift)(f5)	(System) (Shift)(F5)
CONTINUE	(Continue)	(System) (f2)	(System) (f2)	(System) (F2)
DELETE CHARACTER	(Del Chr)	(Del Chr)	(Del)	(Delete)
DELETE LINE	(Del Ln)	(Del Ln)	(Shift)(DEL)	(Shift)(Delete)
DISPLAY FUNCTIONS	(Display Fctns)	(System) (f6)	(System) (f6)	(System) (F6)
DUMP ALPHA	(Dump Alpha)	(Shift) unlabeled key2	(Shift)(F7) or (Shift)(*) (numpad)	(Shift)(F11)
DUMP GRAPHICS	(Dump Graphics)	(Shift) unlabeled key3	(Shift)(F8)	(Shift)(F12)
EDIT	(Edit)	(user 1) (f1)	(user 1) (f1)	(user 1) (F1)
EXECUTE	(EXECUTE)	†	†	†

* Background function. No equivalent in series 200/300.

† Cannot generate this keystroke from this keyboard. If this character is OUTPUT to the keyboard, an error is *not* reported. Instead, the system will perform as much of the indicated action as possible.

‡ Exit to MS-DOS. No equivalent in series 200/300.

Function	HP 98203A,B,C Series 200	HP 46021A ITF Series 300	Vectra PC Keyboard	Enhanced Vectra PC Keyboard
EXIT HP BASIC	‡	‡	(CTRL)(F10) ‡	(CTRL)(F10) ‡
SOFTKEY 0	(k0)	†	†	†
SOFTKEY 1	(k1)	(f1)	(f1)	(F1)
SOFTKEY 2	(k2)	(f2)	(f2)	(F2)
SOFTKEY 3	(k3)	(f3)	(f3)	(F3)
SOFTKEY 4	(k4)	(f4)	(f4)	(F4)
SOFTKEY 5	(k5)	(f5)	(f5)	(F5)
SOFTKEY 6	(k6)	(f6)	(f6)	(F6)
SOFTKEY 7	(k7)	(f7)	(f7)	(F7)
SOFTKEY 8	(k8)	(f8)	(f8)	(F8)
SOFTKEY 9	(k9)	†	†	†
GRAPHICS	(Graphics)	Unlabeled Key3	(F8)	(F12)
INSERT CHARACTER	(Ins Chr)	(Ins Chr)	(Ins)	(Insert)
INSERT LINE	(Ins Ln)	(Ins Ln)	(Shift)(Ins)	(Shift)(Insert)
MENU	†	(Menu)	(F9)	(F9)
PAUSE	(Pause)	(Stop)	(F3)	(Pause)
PRINT ALL	(Prt All)	(System) (f4)	(System) (f4) or (Shift)(F5)	(System) (f4) or (Print Screen)
RECALL	(Recall)	Unlableled Key1	(F2) or (System) (f8) or (Shift)(Pg Up)	(System) (F8) or (Shift)(Page Up)
RECALL FORWARD	(Shift Recall)	(Shift) Unlabeled Key1	(Shift)(F2) or (Shift)(Pg Dn) or (System) (Shift)(f8)	(Shift)(Page Down) or (System) (Shift)(F8)
RESET	(Reset)	(Shift)(Break)	(Shift)(F1)	(Shift)(Scroll Lock)
RESULT	(Result)	Unlabeled Key4	(F4)	RES (keyword)

* Background function. No equivalent in series 200/300.

† Cannot generate this keystroke from this keyboard. If this character is OUTPUT to the keyboard, an error is *not* reported. Instead, the system will perform as much of the indicated action as possible.

‡ Exit to MS-DOS. No equivalent in series 200/300.

Function	HP 98203A,B,C Series 200	HP 46021A ITF Series 300	Vectra PC Keyboard	Enhanced Vectra PC Keyboard
RUN	(Run)	(System) (f3)	(System) (f3)	(System) (F3)
SET TAB	(Set Tab)	(System) (f5)	(System) (f5)	(System) (F5)
SELECT	†	(Select)	(Shift)(Enter)	(Shift)(Enter)
STEP	(Step)	(System) (f1)	(F5) or (System) (f1)	(System) (F1) or (Shift)(Print Screen)
STOP	(Stop)	(Shift)(Stop)	(Shift)(F3)	(Shift)(Pause)
SYSTEM KEYS	†	(System)	(F10)	(F10)
USER KEYS	†	(User)	(Shift)(F10)	(Shift)(F10)

* Background function. No equivalent in series 200/300.

† Cannot generate this keystroke from this keyboard. If this character is OUTPUT to the keyboard, an error is *not* reported. Instead, the system will perform as much of the indicated action as possible.

‡ Exit to MS-DOS. No equivalent in series 200/300.

F   HP BASIC Configuration

When you boot HP BASIC, a configuration file is used to tell the system what resources are available to your computer and how they are to be used. The filename HPW.CON is reserved for the configuration file. HPW.CON must be in the same directory as the boot program, or it must be specified in the MS-DOS search path.

If your configuration file is not correct you may not be able to boot your HPBASIC system.

The configuration file tells the PC emulator which HP Series 200 computer you want to emulate (HP 9816B, HP 9836A, or HP 9836C) and the video mode to use. In addition, the file can specify settings for HP-IB, serial interface cards, and a number of other settings for your HP BASIC system.

The HPW.CON configuration file found on disc one contains the default configuration. This configuration file will work with most PC applications, however you may wish to create a custom configuration that matches your needs. You can do this with the utility program CONF.

Working With the Configuration Utility Program

To use the configuration utility program, first enter MS-DOS. Then make sure you are in the disc drive and directory that contains the HPW.CON file.

- For a system without a hard disc: the HPW.CON file will be in the root directory of disc one. Your MS-DOS prompt should be A:>.

- For a system with a hard disc: the HPW.CON file will most likely be in the directory C:\HPW. Your MS-DOS prompt should be C:\HPW>.

You can execute the configuration utility program with an optional parameter specifying a configuration file to use instead of HPW.CON. If no file is specified, HPW.CON is automatically used.

To execute the configuration utility program without the optional file name, insert disc one of your HP BASIC set of discs into drive A, type:

A:CONF

and press (Enter).

HPW.CON will be used since no file is specified.

To execute the configuration utility program using an optional configuration file, insert disc one of your HP BASIC set of discs into drive A, type:

A:CONF C:\HPW\NEWCON

and press (Enter).

F

In this case, the configuration file used will be a file named C:\HPW\NEWCON.

Note 👆

Before C:\HPW\NEWCON can be used as the configuration file it *must* be renamed as HPW.CON. HP BASIC will *only* recognize HPW.CON as the configuration file.

Once the configuration utility program is running, you will see "Language Processor Configuration Utility rev. 1.5" on the screen. Note how the screen is divided into four areas. On the left is a column labeled "Primary Configurations." On the upper right is the "Serial Cards" area. On the lower right is the "HP-IB Cards" area. Along the bottom are labeled function keys.

Function keys (f1) and (f2) (NEXT CHOICE and PREV CHOICE) will show you the different choices of values for each item in the configuration screen. Function key (f5) (SAVE CONFIG) is used when you have completed changes to the configuration utility program and wish to save the information into the HPW.CON file. Function key (f7) (DEFAULT VALUE) will set the current field (the field where the cursor is currently located) to its default value. Function key (f8) (EXIT) will return you to MS-DOS.

Use the keyboard cursor control keys to move from one item to the next on the screen.

Primary Configurations

Each item in the Primary Configurations area is explained below. As you proceed through the items on the screen, read each explanation to find if you need to change the value. Use the function keys to locate the appropriate value for your system setup.

The following items are covered in the Primary Configuration area:

1. Port address.
2. PC interrupt.
3. Machine type.
4. VGA enable.
5. Background mode.
6. DOS cmd save mode.
7. DOS command wait.
8. Mouse sensitivity.
9. Cache.
10. Drives.
11. Keydefs.

Port Address

Port address is the memory location used by the PC emulator to communicate with the language processor card. Hexidecimal values for the port address range from 0×000 to $0 \times 3F0$ in steps of 0×008. The default value is 0×250. If you change the port address, you will have to change some switch settings for switch SW2 on the language processor card. Refer to the chart on the following page.

F

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

Interface Enable ON

HP-IB System Controller
OFF = Sys Controller
ON = Not

Address Lines

9
8
7
6
5
4
3

ON = 0
OFF = 1

2
1
0
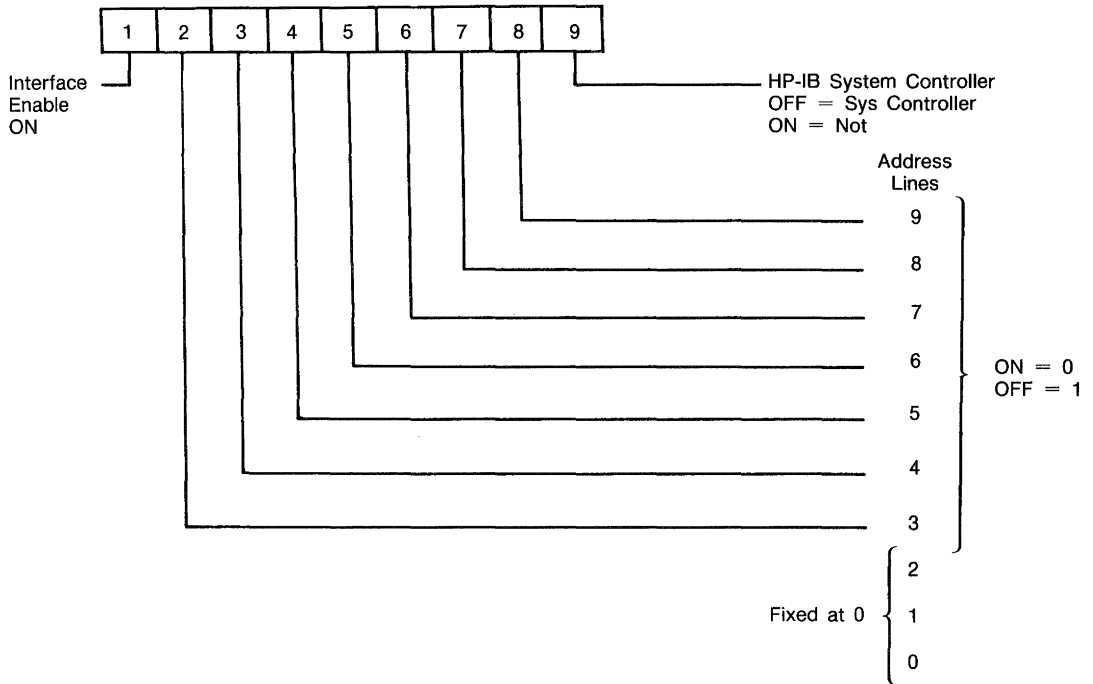
Fixed at 0

Switch SW2 Settings

As shown above, switch segments 2 through 8 determine the port address of the language processor. The last three bits of the address are fixed at 0 and are not set by the switch segments. The switch segments are arranged in reverse order of significant bits, with switch segment 8 representing the most significant bit and switch segment 2 representing the least significant bit. You determine the desired port address in hexadecimal, convert this value to binary, and set the appropriate switch segments accordingly. For example, assume you want the hexadecimal port address to be 0×338. The binary value of 0×338 is 11 0011 1000. Since the last three bits are fixed at 0 and not affected by the switch, they may be disregarded. The switch segment settings are then:

Binary Value	1	1	0	0	1	1	1
SW 2 Setting	8 off	7 off	6 on	5 on	4 off	3 off	2 off

Note that a binary value of 0 corresponds to a switch segment setting of ON, and a binary value of 1 corresponds to a switch segment setting of OFF.

PC Interrupt

PC interrupt changes the PC interrupt level of your language processor card. The default interrupt level is set at IRQ7. If you must change the interrupt level, be sure the level you choose does not conflict with any other cards or devices connected to your system. Alternate values for the PC interrupt level are IRQ3, IRQ4, IRQ5, and IRQ9. The interrupt level you select must also be set on the language processor card.

Machine Type

Machine type is used to select the series 200 display you want to emulate. Use the Series 200 model number to specify the display you want to emulate. Each display has its own number code. You must also indicate whether you would like to emulate separate alpha/graphics modes or combined alpha/graphics modes. When using combined alpha/graphics modes, some performance is lost while displaying alpha text. For best alpha performance, select the separate alpha/graphics mode. If you use the field entry DEFAULT MODE in the HP BASIC configuration file, HP BASIC determines the best emulation mode to use based on the existing video adapter in your PC. The default Machine type values for the various adapters are:

Monochrome Plus:	9816 COMBINED
Multimode:	9836A COMBINED
Enhanced Graphics (EGA):	9836C COMBINED

For more information on displays, refer to appendix B.

VGA Enable

You should set VGA enable to ON when you install a VGA type display interface and a monitor capable of handling the VGA's high resolution mode (640 × 480 dots).

Background Mode

Background mode is used when you want the language processor to continue processing while you leave BASIC and perform other tasks. The language processor will continue working in one of two ways.

- When IGNORE GRAPHICS (continue when graphics is accessed) is selected, the language processor will continue running even if it is supposed to access the graphics display during its work. A graphics display access will be ignored. Therefore, any information that was supposed to be delivered to the graphics display will be lost. The alpha display is updated and maintained while the system is in background mode. The alpha display will be restored when HP BASIC is reentered.

- When PAUSE ON DISPLAY (wait when display is accessed) is selected, the language processor will pause if a display access is made. This is true for both alpha and graphics displays. When the background condition is removed (that is, when you return to HP BASIC), the language processor will display its information on the screen and then continue the operation it was doing at the time it paused.

For more information on Background mode, refer to appendix B.

DOS Cmd Save Mode DOS Command Wait

When you are working in HP BASIC and need to access MS-DOS, you will use the command "OUTPUT 19". "DOS cmd save mode" and "DOS command wait" affect computer behavior when you use "OUTPUT 19."

"DOS cmd save mode" has two possible values, ON and OFF.

- ON will provide a display for MS-DOS command output, but will clear the HP BASIC graphics display. HP BASIC alpha information will be retained.
- OFF will allow MS-DOS to write into the HP BASIC display, resulting in combined output. However, since the HP BASIC graphics display is not cleared in this mode, both HP BASIC alpha and graphics displays are retained and will be intact *provided* MS-DOS does not produce any display output.

"DOS command wait" has two values, ON and OFF. ON causes the computer to pause upon completion of an MS-DOS command. When a key is pressed, the computer will return to HP BASIC. OFF causes the computer to continue processing immediately.

Refer to "The MS-DOS Communications Port" in appendix B for more information.

Mouse Sensitivity

"Mouse sensitivity" is only used if you have a non-HP-HIL mouse and the Microsoft mouse interface standard (INT 33H). The value chosen sets the ratio of mouse movement "ticks" to screen pixel movement.

Cache

"Cache" is a buffer that can be used to store the equivalent of one disc sector of data. Cache can be either ON or OFF, and the default is OFF. Setting cache to ON may improve the speed of writing to disc data files.

When cache is ON, the buffer will store data until it is full. The data is then written onto a sector of the disc, and the buffer is cleared for more data. This method of saving data can be significantly faster than the when cache is OFF. However, if there is a power interruption, or a disc is removed, or the computer is rebooted before closing the file, then all of the data in cache will be lost.

Drives

The HP BASIC Language Processor can have 15 different path names specified for storage of information. The path names are written just as they would be in MS-DOS. First, a drive name is specified and then, if desired, a directory (and possibly a sub-directory). Network devices and other forms of remote or peripheral devices may be included in drive specifications.

If you are using the Hierarchical File System, you *must* specify a partition in addition to the drive name. Legal values for partitions are 2, 3, and 4. The partition then becomes a part of the drive name. For example, C2: is the second partition of drive C. Note that the colon is required following the partition number. Also, directories cannot be used with partitions.

When you select "drives" on the configuration screen, function key (f1) is labeled EDIT DRIVES. Press (f1) and a window will appear in the middle of the screen. You will type the path names into this window.

ONLY ONE path name can be placed on each line in the window. Up to 15 lines may be used. The first path listed will be called "drive 0" by the HP BASIC Language Processor. The second path will be "drive 1," the third will be "drive 2," etc.

For each physical disc drive, the drive list may include a drive specifier (C:) *or* up to 15 directories for that drive (C:\, C:\HPW, etc.), but not both. Note that if a drive specifier (C:) is used, HP BASIC will access the directory that was the current MS-DOS directory for that drive when HP BASIC was executed.

To type path names, use the keyboard typewriter keys. Use the (Ins) key to toggle between Insert and Replace modes.

In Insert mode, characters are inserted before the character the cursor is currently on. The cursor is a block cursor, and the backspace key deletes characters.

In Replace mode, the current character is replaced by the new character. The cursor is an underscore, and the backspace key acts the same as a left cursor control key.

The (DEL) key deletes the current character.

If you need more help in understanding how to specify MS-DOS path names, refer to an MS-DOS reference manual.

When you finish correctly typing path names into the window, press (f8) (Return to MAIN) to return to the main screen again. Notice that there is a number next to the "drives" item. This number corresponds to the number of paths you have entered.

Keydefs

"Keydefs" is used to redefine the function (series 200/300 keystroke) attached to a PC keystroke. Use "keydefs" if you feel you can make the keyboard more convenient for your personal use.

When you select "keydefs" on the configuration screen, function key (f1) is labeled EDIT KEYS. Press (f1) and a "window" appears in the middle of the screen. A vertical line divides the window. The left side of the window contains a hidden list of options for Series 200/300 keyboards. The right side of the window contains a hidden list of options for PC keyboards. The options are revealed one at a time when you use (f1) (NEXT CHOICE) and (f2) (PREV CHOICE). As you step through the choices, you will see the key function and, in the right margin, a 1 or 2-letter code. On the left side of the window, the codes will be "N", "I", or both. "N" means the key is found on the Series 200 keyboards, and "I" means the key is

found on the the ITF (Series 300) keyboards. On the right side of the window, the codes will be "E", "V", or both. "E" means the key is present on the Enhanced Vectra PC keyboard, and "V" means the key is present on the Vectra PC keyboard.

To illustrate the use of "keydefs", let's assume you want to change the keystroke for clearing a line when in BASIC. On a Series 200 keyboard, there is a key labeled (CLR LN) (clear line). Using (f1) and (f2), locate this keystroke in the hidden list of options on the left side of the window. Move the cursor to the right side of the window (on the same line). Assume that you would prefer to have (Ctrl)(l) be the keystroke for clearing the line. First, check the table of keystrokes in appendix E to make sure this keystroke hasn't already been used. Second, press (f4) to indicate (Ctrl). Third, use (f1) (NEXT CHOICE) and (f2) (PREV CHOICE) to locate the (l) key.

When you finish correctly setting keystrokes into the window, press (f8) (Return to MAIN) to return to the main screen again. Notice that there is a number next to the "keydefs" item. This number corresponds to the number of "keydef" changes you entered.

Serial Cards

Each item in the Serial Cards area is explained below. As you proceed through the items on the screen, read each explanation to help you determine if you need to change the value. Use the function keys to locate the appropriate value for your system setup.

You may configure up to two serial cards in a single configuration file. There are two columns of input spaces next to the list of items. Use one column for each serial card.

The following items are covered in this section:

1. Com port.

2. Sel code.

3. Int level.

4. Baud, parity, char len, and stop bits.

5. Modem disconnect.

Com Port

"Com port" allows you to select which of your serial ports to configure, COM1 or COM2. The port you select will refer to the serial uinterface with that switch setting. You may also specify NONE (no port), in which case the rest of the settings are not displayed.

Sel Code

"Sel code" specifies the select code used by HP BASIC to access the serial interface. The select code value can be either 9 or 23.

Int Level

"Int level" is used to set the HP BASIC interrupt level for the serial interface. The available values are 3, 4, 5, and 6. This interrupt level may be shared with other devices.

F

Baud, Parity, Char Len, and Stop Bits

These four items specify information about communication between HP BASIC and a serial device (such as a plotter or printer). Documentation that came with your serial device will state what the proper values should be.

Modem Disconnect

There are four items in modem disconnect. The "on" choices are CD (carrier detect), RI (ring indicator), DSR (data set ready), and CTS (clear to send). The "off" choices are ––, ––, –––, and –––. (The number of hyphens matches the number of letters in the "on" position code.) Documentation that came with your modem will help you determine the proper values to use.

HP-IB Cards

Each item in the HP-IB Cards area is explained below. As you proceed through the items on the screen, read each explanation to find if you need to change the value. Use the function keys to locate the appropriate value for your system setup.

You may have up to two HP-IB cards installed. There are two columns of input spaces next to the list of items. Use one column for each HP-IB card.

The following items are covered in this section:

1. PC sel code.
2. PC int level.
3. Sel code.
4. Int level.

PC Sel Code

"PC sel code" is used by the PC emulator to access the HP-IB interface. This is the number you would use to access the interface from MS-DOS. It must match the switch select code set on the HP 82990 HP-IB card. The available values are NONE, and the numbers 1 through 16.

PC Int Level

"PC int level" sets the PC interrupt level of the HP-IB card. The available values are NONE, 3, 4, 5, and 6. If you enable interrupts, be sure the level you choose does not conflict with any other cards or devices connected to your system. Note that COM1 (if available) uses PC interrupt level 4 and COM2 (if available) uses PC interrupt level 3. HP-IB cards do not use interrupts to access mass storage devices, therefore you may select NONE (interrupt disabled) if you intend to use HP-IB to access only mass storage devices.

Sel Code   "Sel code" is used by HP BASIC to access the HP-IB interface. The value for select code can be either 24 or 25.

Int Level   "Int level" indicates the HP BASIC interrupt level of the interface. The available values are 3, 4, 5, and 6. These interrupt levels may be shared with other devices. For example, two HP-IB cards may be on HP BASIC interrupt level 3, but only one can be on *PC* interrupt level 3.

Exiting the Configuration Utility Program

When you have finished making changes in the configuration utility program, be sure you save the changes you made by pressing (f5) (SAVE CONFIG) before you exit.

After saving the changes, press (f8) (EXIT) to return to MS-DOS.

F

G

List of Example Program Files

This appendix lists all the example program files on your Manual Examples disc. These programs are referred to in chapters 3, 4, and 5.

Chapter 3

INDNTPGM
REPEAT1
WHILE1
ONKEY1
DROUND1
MATSORT
CRBDAT
OUTPUT1
ONCYCLE
ONDELAY
PRIORITY
TRACEALL
TRPAUSE

Chapter 4

BOLT
BIGLINES
SCALE
SCALE2
SCALE3
SCALE4
LABELS
AXES
STARS
PENDEMO
LINETYPES
PDIRDEMO

G

H   Error Messages

1   Missing option or configuration error. If a statement requires an option which is not loaded, the option number or option name is given along with the error number.

2   Memory overflow. If you get this error while loading a file, the program is too large for the computer's memory. If the program loads, but you get the error when you press (Run), then the overflow was caused by the variable declarations. Either way, you need to modify the program or add more memory.

3   Line not found in current context. This could be a GOTO or GOSUB that references a non-existent line, or an EDIT command that refers to a non-existent label.

4   Improper RETURN. Executing a RETURN statement without previously executing an appropriate GOSUB or function call. Also, a RETURN statement in a user-defined function with no value specified.

5   Improper context terminator. You forgot to put an END statement in the program. Also applies to SUBEND and FNEND.

6   Improper FOR...NEXT matching. Executing a NEXT statement without previously executing the matching FOR statement. Indicates improper nesting or overlapping of the loops.

7   Undefined function or subprogram. Attempt to call a SUB or user-defined function that is not in memory. Look out for program lines that assumed an optional CALL.

8 Improper parameter matching. A type mismatch between a pass parameter and a formal parameter of a subprogram.

9 Improper number of parameters. Passing either too few or too many parameters to a subprogram. Applies only to non-optional parameters.

10 String type required. Attempting to return a numeric from a user-   defined string function.

11 Numeric type required. Attempting to return a string from a user-defined numeric function.

12 Attempt to redeclare variable. Including the same variable name twice in declarative statements such as DIM or INTEGER.

13 Array dimensions not specified. Using the (*) symbol after a variable name when that variable has never been declared as an array.

14 OPTION BASE not allowed here. The OPTION BASE statement must appear before any declarative statements such as DIM or INTEGER. Only one OPTION BASE statement is allowed in one context.

15 Invalid bounds. Attempt to declare an array with more than 32767 elements or with upper bound less than lower bound.

16 Improper or inconsistent dimensions. Using the wrong number of subscripts when referencing an array element.

17 Subscript out of range. A subscript in an array reference is outside the current bounds of the array.

18 String overflow or substring error. String overflow is an attempt to put too many characters into a string (exceeding dimensioned length). This can happen in an assignment, an ENTER and INPUT, or a READ. A substring error is an attempted violation of the rules for substrings. Watch out for null strings where you weren't expecting them.

H

19 Improper value or out or range. A value is too large or too small. Applies to items found in a variety of statements. Often occurs when the number bulder overflows (or underflows) during an I/O operation.

20 INTEGER overflow. An assignment or result exceeds the range allowed for INTEGER variables. Must be -32768 thru 32767.

22 REAL overflow. An assignment or result exceeds the range allowed for REAL variables.

24 Trig argument too large for accurate evaluation. Out-of-range argument for a function such as TAN or LDIR.

25 Magnitude of ASN or ACS argument is greater than 1. Arguments to these functions must be in the range -1 thru $+1$.

26 Zero to non-positive power. Exponentiation error.

27 Negative base to non-integer power. Exponentiation error.

28 LOG or LGT of a non-positive number.

29 Illegal floating point number. Does not occur as a result of any calculations, but is possible when a FORMAT OFF I/O operation fills a REAL variable with something other than a REAL number.

30 SQR of a negative number.

31 Division (or MOD) by zero.

32 String does not represent a valid number. Attempt to use "non-numeric" characters as an argument for VAL, data for a READ, or in response to an INPUT statement requesting a number.

33 Improper argument for NUM or RPT$. Null string not allowed.

34 Referenced line not an IMAGE statement. A USING clause contains a line identifier, and the line referred to is not an IMAGE statement.

35 Improper image. See IMAGE or the appropriate keyword in appendix A.

36 Out of data in READ. A READ statement is expecting more data than is available in the referenced DATA statements. Check for deleted lines, proper OPTION BASE, proper use of RESTORE, or typing errors.

38 TAB or TABXY not allowed here. The tab functions are not allowed in statements that contain a USING clause. TABXY is allowed only in a PRINT statement.

40 Improper REN, COPYLINES, or MOVELINES command. Line numbers must be whole numbers from 1 to 32766. This may also result from a COPYLINES or MOVELINES statement whose destination line numbers lie within the source range.

41 First line number greater than second line number. Parameters out of order in a statement like SAVE, LIST, or DEL.

43 Matrix must be square. The MAT functions: IDN, INV, and DET require the array to have equal numbers of rows and columns.

44 Result cannot be an operand. Attempt to use a matrix as both result and argument in a MAT TRN or matrix multiplication.

46 Attempting a SAVE when there is no program in memory.

47 COM declarations are inconsistent or incorrect. Includes such things as mismatched dimensions, unspecified dimensions, and blank COM occurring for the first time in a subprogram.

49 Branch destination not found. A statement such as ON ERROR or ON KEY refers to a line that does not exist. Branch destinations must be in the same context as the ON... statement.

51 File not currently assigned. Attempting an ON/OFF END statement with an unassigned I/O path name.

52 Improper mass storage unit specifier. The characters used for a msus do not form a valid specifier. This could be a missing colon, too many parameters, illegal characters, etc.

53 Improper file name. File names are limited to 10 characters. Foreign characters are allowed, but punctuation is not.

54 Duplicate file name. The specified file name already exists in directory. It is illegal to have two files with the same name on one volume.

55 Directory overflow. Although there may be room on the media for the file, there is no room in the directory for another file name. Discs initialized by BASIC have room for over 100 entries in the directory, but other systems may make a directory of a different size.

56 File name is undefined. The specified file name does not exist in the directory. Check the contents of the disc with a CAT command.

58 Improper file type. Many mass storage oerations are limited to certain file types. For example, LOAD is limited to PROG files and ASSIGN is limited to ASCII and BDAT files.

59 End of file or buffer found. For files: No data left when reading a file, or no space left when writing a file. For buffers: No data left for an ENTER, or no buffer space left for an OUTPUT. Also, WORD-mode TRANSFER terminated with odd number of bytes.

60 End of record found in random mode. Attempt to ENTER a field that is larger than a defined record.

62 Protect code violation. Failure to specify the protect code of a protected file, or attempting to protect a file of the wrong type.

64 Mass storage media overflow. There is not enough contiguous free space for the specified file size. The disc is full.

65 Incorrect data type. The array used in a graphics operation, such as GLOAD, is the wrong type (INTEGER or REAL).

66 INITIALIZE failed. Too many bad tracks found. The disc is defective, damaged, or dirty.

67 Illegal mass storage parameter. A mass storage statement contains a parameter that is out of range, such as a negative record number or an out of range number or records.

68 Syntax error occurred during GET. One or more lines in the file could not be stored as valid program lines. The offending lines are usually listed on the system printer. Also occurs if the first line in the file does not start with a valid line number.

72 Disc controller not found or bad controller address. The msus contains an improper device selector, or no external disc is connected.

73 Improper device type in mass storage unit specifier. The msus has the correct general form, but the characters used for a device typer are not recognized.

76 Incorrect unit number in mass storage unit specifier. The msus contains a unit number that does not exist on the specified device.

77 Attempt to purge an open file. The specified file is assigned to an I/O path name which has not been closed.

78 Invalid mass storage volume label. Usually indicates that the media has not been initialized on a compatible system. Could also be a bad disc.

79 File open on target device. Attempt to copy an entire volume with a file open on the destination disc.

80 Disc changed or not in drive. Either there is no disc in the drive or the drive door was opened while a file was assigned.

If you are using a floppy disc on your HP BASIC system for the first time, be sure the disc is *not* write protected. HP BASIC checks for the existence of the directory file HPWLIF.DIR. If this file is not found on the disc, HP BASIC will create it. The error results when HP BASIC tries to write the newly created directory file to the write-protected disc.

H

81 Mass storage hardware failure. Also occurs when the disc is pinched and not turning. Try reinserting the disc.

82 Mass storage unit not present.

83 Write protected. Attempting to write to a write-protected disc. This includes many operations such as PURGE, INITIALIZE, CREATE, SAVE, OUTPUT, etc.

84 Record not found. Usually indicates that the media has not been initialized, or there is a problem in the HPW file system. Refer to appendix C.

When initializing a disc, an Error 84 (record not found) can occur under either of the following conditions:

- HP BASIC cannot find the MS-DOS FORMAT program.
- There is not enough PC RAM to load the FORMAT program.

You can insure that MS-DOS can find the FORMAT program by specifying the proper path on a PATH statement in your AUTOEXEC.BAT file. If an Error 84 occurs and you suspect insufficient memory to be the problem, you may have to format the disc directly from MS-DOS by using the FORMAT command.

85 Media not initialized. (Usually not produced by the internal drive.)

87 Record address error. Usually indicates a problem with the media.

88 Read data error. The media is physically or magnetically damaged, and the data cannot be read.

89 Checkread error. An error was detected when reading the data just written. The media is probably damaged.

90 Mass storage system error. Usually a problem with the hardware or the media.

93 Incorrect volume code in MSUS. The MSUS contains a volume number that does not exist on the specified device.

100 Numeric IMAGE for string item.

101 String IMAGE for numeric item.

102 Numeric field specifier is too large. Specifying more than 256 characters in a numeric field.

103 Item has no corresponding IMAGE. The image specifier has no fields that are used for item processing. Specifiers such as # X / are not used to process the data for the item list. Item-processing specifiers include things like K D B A.

105 Numeric IMAGE field too small. Not enough characters are specified to represent the number.

106 IMAGE exponent field too small. Not enough exponent characters are specified to represent the number.

107 IMAGE sign specifier missing. Not enough characters are specified to represent the number. Number would fit except for the minus sign.

117 Too many nested structures. The nesting level is too deep for such structures as FOR, SELECT, IF, LOOP, etc.

118 Too many structures in context. Refers to such structures as FOR/NEXT, IF/THEN/ELSE, SELECT/CASE, WHILE, etc.

120 Not allowed while program running. The program must be stopped before you can execute this command.

121 Line not in main program. The run line specified in a LOAD or GET is not in the main context.

122 Program is not continuable. The program is in the stopped state, not the paused state. CONT is allowed only in the paused state.

126 Quote mark in unquoted string. Quote marks must be used in pairs.

127 Statements which affect the knob mode are out of order.

128 Line too long during GET.

131 Unrecognized non-ASCII keycode. An output to the keyboard contained a CHR$(255) followed by an illegal byte.

132 Keycode buffer overflow. Trying to send too many characters to the keyboard buffer with an OUTPUT 2 statement.

133 DELSUB of non-existent or busy subprogram.

134 Improper SCRATCH statement.

135 READIO/WRITEIO to nonexistent memory location.

136 REAL underflow. The input or result is closer to zero than 10^{-308} (approximately).

140 Too many symbols in the program. Symbols are variable names, I/O path names, COM block names, subprogram names, and line identifiers.

141 Variable cannot be allocated. It is already allocated.

142 Variable not allocated. Attempt to DEALLOCATE a variable that was not allocated.

143 Reference to missing OPTIONAL parameter. The subprogram is trying to use an optional parameter that didn't have any value passed to it. Use NPAR to check the number of passed parameters.

145 May not build COM at this time. Attempt to add or change COM when a program is running. For example, a program does a LOADSUB and the COM in the new subprogram does not match existing COM.

146 Duplicate line label in context. There cannot be two lines with the same line label in one context.

150 Illegal interface select code or device selector. Value out of range.

152 Parity error.

153 Insufficient data for ENTER. A statement terminator was received before the variable list was satisfied.

154 String greater than 32767 bytes in ENTER.

155 Improper interface register number. Value out of range or negative.

156 Illegal expression type in list. For example, trying to ENTER into a constant.

157 No ENTER terminator found. The variable list has been satisfied, but no statement terminator was received in the next 256 characters. The # specifier allows the statement to terminate when the last item is satisfied.

158 Improper image specifier or nesting images more than 8 deep. The characters used for an image specifier are improper or in an improper order.

159 Numeric data not received. When entering characters for a numeric field, an item terminator was encountered before any "numeric" characters were received.

160 Attempt to enter more than 32767 digits into one number.

163 Interface not present. The intended interface is not present, set to a different select code, or is malfunctioning.

164 Illegal BYTE/WORD operation. Attempt to ASSIGN with the WORD attribute to a non-word device.

165 Image specifier greater than dimensioned string length.

167 Interface status error. Exact meaning depends upon the interface type. With HP-IB, this can happen when a non-controller operation by the computer is aborted by the bus. It may also indicate a problem in the HPW file system. Refer to appendix C.

168 Device timeout occurred and the ON TIMEOUT branch could not be taken.

170 I/O operation not allowed. The I/O statement has the proper form, but its operation is not defined for the specified device. For example, using an HP-IB statement on a non-HP-IB interface or directing a LIST to the keyboard.

H

171 Illegal I/O addressing sequence. The secondary addressing in a device selector is improper or primary address too large for specified device.

172 Peripheral error. PSTS line is false. If used, this means that the peripheral device is down. If PSTS is not being used, this error can be suppressed by using control register 2 of the GPIO.

173 Active or system controller required. The HP-IB is not active controller and needs to be for the specified operation.

174 Nested I/O prohibited. An I/O statement contains a user-defined function. Both the original statement and the function are trying to access the same file or device.

177 Undefined I/O path name. Attempting to use an I/O path name that is not assigned to a device or file.

178 Trailing punctuation in ENTER. The trailing comma or semicolon that is sometimes used at the end of OUTPUT statements is not allowed at the end of ENTER statements.

301 Cannot do while connected.

303 Not allowed when trace active.

304 Too many characters without terminator.

306 Interface card failure. The datacomm card has failed self-test.

308 Illegal character in data. Datacomm error.

310 Not connected. Datacomm error.

313 USART receive buffer overflow. Overrun error detected. Interface card is unable to keep up with incoming data rate. Data has been lost.

314 Receive buffer overflow. Program is not accepting data fast enough to keep up with incoming data rate. Data has been lost.

315 Missing data transmit clock. A transmit timeout has occurred because a missing data clock prevented the card from transmitting. The card has disconnected from the line.

316 CTS false too long. The interface card was unable to transmit for a predetermined period of time because Clear-To-Send was false on a half-duplex line. The card has disconnected from the line.

317 Lost carrier disconnect. Data Set Ready (DSR) or Data Carrier Detect (if full duplex) went inactive for too long.

318 No activity disconnect. The card has disconnected from the line because no data was transmitted or received for a predetermined length of time.

319 Connection not established. Data Set Ready or Data Carrier Detect (if full duplex) did not become active within a predetermined length of time.

324 Card trace buffer overflow.

325 Illegal databits/parity combination. Attempting to program 8 bits-per-character and a parity of "1" or "0".

326 Register address out of range. A control or status register access was attempted to a non-existent register.

327 Register value out of range. Attempting to place an illegal value in a control register.

328 USART Transmit underrun.

330 User-defined LEXICAL ORDER IS table size exceeds array size.

331 Repeated value in pointer. A MAT REORDER vector has repeated subscripts. This error is not always caught.

332 Non-existent dimension given. Attempt to specify a non-existent dimension in a MAT REORDER operation.

333 Improper subscript in pointer. A MAT REORDER vector specifies a non-existent subscript.

H

334 Pointer size is not equal to the number of records. A MAT REORDER vector has a different number of elements than the specified dimension of the array.

335 Pointer is not a vector. Only single-dimension arrays (vectors) can be used as the pointer in a MAT REORDER or a MAT SORT statement.

337 Substring key is out of range. The specified substring range of the sort key exceeds the dimensioned length of the elements in the array.

338 Key subscript out of range. Attempt to specify a subscript in a sort key outside the current bounds of the array.

340 Mode table too long. User-defined LEXICAL ORDER IS mode table contains more than 63 entries.

341 Improper mode indicator. User-defined LEXICAL ORDER IS table contains an illegal combination of mode type and mode pointer.

342 Not a single-dimension integer array. User-defined LEXICAL ORDER IS mode table must be a single-dimension array of type INTEGER.

343 Mode pointer is out of range. User-defined LEXICAL ORDER IS table has a mode pointer greater than the existing mode table size.

344 1 for 2 list empty or too long. A user-defined LEXICAL ORDER IS table contains an entry indicating an improper number of 1 for 2 secondaries.

345 CASE expression type mismatch. The SELECT statement and its CASE statements must refer to the same general type, numeric or string.

346 INDENT parameter out of range. The parameters must be in the range: 0 thru eight characters less than the screen width.

347 Structures improperly matched. There is not a corresponding number of structure beginnings and endings. Usually means that you forgot a statement such as END IF, NEXT, END SELECT, etc.

349 CSUB has been modified. A contiguous block of compiled subroutines has been modified since it was loaded. A single module that shos as multiple CSUB statements has been altered because program lines were inserted or deleted.

353 Data link failure.

369 **-399** Errors in this range are reported if a run-time Pascal error occurs in a CSUB. To determine the Pascal error number, subtract 400 from the BASIC error number. Information on the Pascal error can be found in the *Pascal Workstation System* manual.

401 Bad system function argument. An invalid argument was given to a time, date, base conversion, or SYSTEMS$ function.

403 Copy failed; program modification incomplete. An error occurred during a COPYLINES or MOVELINES resulting in an incomplete operation. Some lines may not have been copied or moved.

427 Priorty may not be lowered.

450 Volume not found—SRM error.

451 Volume labels do not match—SRM error.

453 File in use—SRM error.

454 Directory formats do not match—SRM error.

455 Possibly corrupt file—SRM error.

456 Unsupporteda directory operation—SRM error.

457 Passwords not supported—SRM error.

458 Unsupported directory format—SRM error.

459 Specified file is not a directory—SRM error.

460 Directory not empty—SRM error.

462 Invalid password—SRM error.

H

465   Invalid rename across volumes—SRM error.

471   TRANSFER not supported by the interface.

481   File locked or open exclusively—SRM error.

482   Cannot move a directory with a RENAME operation—SRM error.

483   System down—SRM error.

484   Password not found—SRM error.

485   Invalid volume copy—SRM error.

488   DMA hardware required. HP 9885 disc drive requires a DMA card or is malfunctioning.

511   The result arary in a MAT INV must be of type REAL.

600   Attribut cannot be modified. The WORD/BYTE mode cannot be changed after assigning the I/O path name.

601   Improper CONVERT lifetime. When the CONVERT attribute is included in the assignment of an I/O path name, the name of a string variable containing the conversion is also specified. The conversion string must exist as long as the I/O path name is valid.

602   Improper BUFFER lifetime. The variable designated as a buffer during an I/O path name assignment must exist as long as the I/O path name is valid.

603   Variable was not declared as a BUFFER. Attempt to assign a variable as a buffer without first declaring the variable as a BUFFER.

604   Bad source or destination for a TRANSFER statement. Transfers are not allowed to the CRT, keyboard, or tape backup on CS80 drives. Buffer to buffer or device to device transfers are not allowed.

605   BDAT file type required. Only BDAT files can be used in a TRANSFER operation.

606   Improper TRANSFER parameters. Conflicting or invalid TRANSFER parameters were specified, such as RECORDS without and EOR clause, or DELIM with an outbound TRANSFER.

607   Inconsistent attributes. Such as CONVERT or PARITY with FORMAT OFF.

609   IVAL or DVAL result too large. Attempt to convert a binary, octal, decimal, or hexadecimal string into a value outside the range of the function.

612   BUFFER pointers in use. Attempt to change one or more buffer pointers while a TRANSFER is in progress.

700   Improper plotter specifier. The characters used as a plotter specifier are not recognized. May be misspelled or contain illegal characters.

702   CRT graphics hardware missing. Hardware problem.

704   Upper bound not greater than lower bound. Applies to P2 < =P1 or VIEWPORT upper bound and CLIP limits.

705   VIEWPORT or CLIP beyond hard clip limits.

708   Device not initialized.

713   Request not supported by specified device. Trying to equate color CRT characteristics with an external device, such as using COLOR MAP on a plotter.

733   GESCAPE opcode not recognized. Only values 1 thru 5 can be used.

900   Undefined typing aid key.

901   Typing aid memory overflow.

902   Must delete entire context. Attempt to delete a SUB or DEF FN statement without deleting its entire context. Easiest way to delete is with DELSUB.

903   No room to renumber. While EDIT mode was renumbering during an insert, all available line numbers were used betweeen insert location and end of program.

904   Null FIND or CHANGE string.

905   CHANGE would produce a line too long for the system. Maximum line length is two lines on the CRT.

906 SUB or DEF FN not allowed here. Attempt to insert a SUB or DEF FN statement into the middle of a context. Subprograms must be appended at the end.

909 May not replace SUB or DEF FN. Similar to deleting a SUB or DEF FN.

910 Identifier not found in this context. The keyboard-specified variable does not already exist in the program. Variables cannot be created from the keyboard; they must be created by running a program.

911 Improper I/O list.

920 Numeric constant not allowed.

921 Numeric identifier not allowed.

922 Numeric array element not allowed.

923 Numeric expression not allowed.

924 Quoted string not allowed.

925 String identifier not allowed.

926 String array element not allowed.

927 Substring not allowed.

928 String expression not allowed.

929 I/O path name not allowed.

930 Numeric array not allowed.

931 String array not allowed.

932 Excess keys specified. A sort key was specified following a key which specified the entire record.

935 Identifier is too long: 15 characters maximum.

936 Unrecognized character. Attempt to store a program line containing an improper name or illegal character.

937 Invalid OPTION BASE. Only 0 and 1 are allowed.

939 OPTIONAL appears twice. A parameter list may have only one OPTIONAL keyword. All parameters listed before it are required, all listed after it are optional.

940   Duplicate formal parameter name.

942   Invalid I/O path name. The characters after the @ are not a valid name. Names must start with a letter.

943   Invalid function name. The characters after the FN are not a valid name. Names must start with a letter.

946   Dimensions are inconsistent with previous declaration. The references to an array contain a different number of subscripts at different places in the program.

947   Invalid array bounds. Value out of range, or more than 32767 elements specified.

948   Multiple assignment prohibited. You cannot assign the same value to multiple variables by stating $X=Y=Z=0$. A separate assignment must be made for each variable.

949   This symbol not allowed here. This is the general "syntax error" message. The statement you typed contains elements that don't belong together, are in the wrong order, or are misspelled.

950   Must be a positive integer.

951   Incomplete statement. This keyword must be followed by other items to make a valid statement.

961   CASE expression type mismatch. The CASE line contains items that are not the same general type, numeric or string.

962   Programmable only: cannot be executed from the keyboard.

963   Command only: cannot be stored as a program line.

977   Statement is too complex. Contains too many operators and functions. Break the expression down so that it is performed by two or more program lines.

980   Too many sysmbols in this context. Symbols include variable names, I/O path names, COM block names, subprogram names, and line identifiers.

982   Too many subscripts: maximum of six dimensions allowed.

H

983 Wrong type or number of parameters. An improper parameter list for a machine-resident function.

985 Invalid quoted string.

987 Invalid line number: must be a whole number 1 thru 32766.

988 This error can be generated by using a configuration file different from the one that was accessed when HP BASIC was booted. If you change the HP BASIC configuration file, you must reboot your HP BASIC system.

Index

ATN, 3-48
ATNH, 3-48
Asynchronous data communication, 5-42
AXES, 4-17

B

Background, 1-3, B-10
Background value, 4-62
Backplane, 5-2
Bar code reader, 5-74
BASE, 3-56
Base conversion, 3-101
Base conversion functions, 3-55
BASIC, 2-12
BASIC, using with SRM, 6-7
Baud rate, serial interface, 5-48
BDAT, 3-129, 3-130, 3-138 thru 3-145, 3-149
BDAT file, reading and writing, 3-138
BINAND, 3-49
Binaries, 1-3, 2-9, 2-11, 3-23, D-1, D-2
Binary functions, 3-48
BINCMP, 3-49
BINEOR, 3-49
BINIOR, 3-49
BIT, 3-49
Boolean,
    arrays, 3-87
    expression, 3-59
Booting,
    from SRM, 6-4
    HP BASIC, 2-8, 2-9, 2-11
Boundary conditions, 3-179
Branching
    conditional, 3-32
    event-initiated, 3-29
    on clock events, 3-173
    restrictions, 3-178
    simple, 3-30
BUFFERS, 2-4
Bus, 5-2
    activity, aborting, 5-28
    sequences, HP-IB, 5-21

C

CALL, 3-116, 3-178
Calling a subprogram, 3-106
Card, language processor, 1-3
CASE, 3-34, 3-35
CASE ELSE, 3-34, 3-36
CAT, 3-20, 3-23, 3-25, 3-150, 3-154, 3-155, 3-157
Catalog, to printer, 3-155
Cataloging select files, 3-155
CHANGE, 3-9
Character format, serial interface, 5-43, 5-47, 5-48
Character sets, selecting, 4-60
Check in a file, C-4
CHR$, 3-161
CLEAR, 5-24, 5-28
CLEAR I/O, 3-169, 3-193, 4-57
Clearing HP-IB devices, 5-28
Clip limits
    hard, 4-14
    soft, 4-14, 4-45
CLIP OFF, 4-14
Clipping, 4-45
ClOCK, 3-178
Clock
    branching on, 3-173
    initial value, 3-170
    range and accuracy, 3-169
    reading, 3-170
    real time, 3-169 thru 3-178
    setting, 3-171
Closing I/O path names, 3-138, 5-15
Color
    background, 4-62
    color-mapped, 4-64
    color-map pen values, 4-65
    default, 4-64
    fill, 4-67
    models, 4-66
    non-color-mapped, 4-62
    graphics, 4-62 thru 4-67
COLORLINE, 4-64

D

DATA, 3-70, 3-125
Data
  storing in programs, 3-123
  writing, 3-142
  driven plotting, 4-47
  file, structure, 3-128
  flow, 5-10
  input, 3-124
  input, serial interface, 5-50
  output, serial interface, 5-50
  pointer, 3-126, 3-128
  storage and retrieval, 3-123
  transfer, serial interface, 5-45, 5-49
  types, numeric, 3-43
Date, 3-170, 3-172
DATE, 3-55
DATE$, 3-170, 3-173
Debugging, 3-184 thru 3-193
Default
  colors, 4-64
  configuration, 2-2 thru 2-4, appendix F
DEF FN, 3-8, 3-104, 3-119, 3-120
Defined records, 3-139
DEG, 3-47, 3-52, 4-42, 4-50
DEL, 3-4
Delays, 3-174
DELETE LINE, 3-4
Deleting lines, 3-4
DET, 3-56
Device selector, 3-19, 3-135, 3-158 thru 3-160,
    5-11, 5-13, 5-17
Device type, 3-19, 3-135
Devices, remote control of, 5-25
Difficulty, SRM 6-32
DIGITIZE, 4-8, 5-72
DIM, 3-45
Dimensioning
  array, 3-62, 3-68
  implicit, 3-68
Directory, 3-18, 3-154, 6-8
Directory paths, 6-8, 6-31
Disc
  initialization, 3-21
  labels, 3-22

DISP, 3-53
Displays,
  alpha, 4-1, B-5 thru B-9
  graphics, 4-1, B-5 thru B-9
Dithering, 4-62, 4-67
DIV, 3-57
DOT, 3-56
DRAW, 4-11
Drawing
  lines, 4-9
  modes, 4-46
DROUND, 3-50, 3-60
DROUND1, 3-60
Dump
  graphics, 4-56
  screen, 4-56
DUMP GRAPHICS, 4-56
Dumping raster images, 4-56
DVAL, 3-55

E

EDGE, 4-27, 4-31, 4-37
EDIT, 3-3
EDIT,
  getting out of, 3-10
  screen, 3-3
Editing
  program, 3-7
  subprograms, 3-120
Editor, 3-2
EIRSERVE, 5-65
Emulator, 1-3
END, 3-29, 3-146
END IF, 3-33
END SELECT, 3-35
Enter, 3-2
ENTER, 3-56, 3-73, 3-129, 3-145, 3-146, 3-154,
    3-155, 5-9, 5-22, 5-58, 6-23, B-13
Entering a program, 3-3
EOF, 3-139, 3-144, 3-145, 3-148, 3-157
EOR, 3-144, 3-148
Erasing lines, 4-22
ERRL, 3-181, 3-183
ERRM$, 3-181
ERRN, 3-181

I

I/O path, 3-142
  closing, 3-138
  opening, 3-137
  names, 5-13
I/O process, 5-7
IDRAW, 4-9, 4-11, 4-21, 4-51
IF...END IF, 3-33
IF...THEN, 3-32
IMAG, 3-53
Image specifier, 3-164, 3-165, 3-167, 3-168
Images, using in printing, 3-163 thru 3-169
IMOVE, 4-7, 4-11, 4-21, 4-51
Implicit dimensioning, 3-66
Incremental plotting, 4-51
INDENT, 3-34
INPUT, 3-82, 3-179
INSTALL, 2-8, 2-10
Indenting, 3-11
INITIALIZE, 3-22, 3-135, 3-136
Initializing a disc, 3-21
Input data, 3-124
INSERT LINE, 3-4
Inserting lines, 3-4
Installing HP BASIC,
  on a hard disc, 2-10
  on a high-capacity floppy disc, 2-5
  source drive, 2-5 thru 2-10
  target drive, 2-5 thru 2-10
INT, 3-47
INTEGER, 3-43, 3-62, 3-135
INTEGER OVERFLOW error, 3-45, 3-49
INTEGER variables, 3-44, 3-66
INTENSITY, 4-62, 4-66
Interface,
  functions, 5-3
  GPIO, 5-6
  HP-IB, 5-4
  select code, 3-158, 5-54
  serial, 5-6
  techniques, 5-1
Interleave, 3-130

Interrupt,
  enable mask, 5-36
  level, 2-4
  priority, GPIO, 5-54
  service routines, GPIO, 5-64
Interrupts, GPIO, 5-62
IPLOT, 4-21, 4-51, 4-67
Isotropic,
  scaling, 4-14
  units, 4-13
IVAL, 3-55
Iterations, fixed number, 3-37

K

Keyboard,
  Enhanced Vectra PC, 3-2, E-22 thru E-43
  functions, 3-2, E-41 thru E-43
  live, 3-15, 3-116, 3-185
  Vectra PC, 3-2, E-4 thru E-21, E-41 thru E-43
Keyword, 3-1

L

LABEL, 4-16, 4-43, 4-60
Labelling a plot, 4-16, 4-37
Labels, softkey, 2-13
LDIR, 4-42
LET, 3-43
Lexical order, 3-101 thru 103
LIF, 3-21, 3-129, B-14 thru B-17
LGT, 3-47
Limit functions, 3-49
Line,
  attributes, 4-22
  label, 3-200
  types, 4-24
  value, 4-62
Linear flow, 3-29
LIST, 3-7, 3-159
LIST BIN, 2-14
Listeners, addressing multiple, 5-22
Listing a program, 3-6

HEWLETT PACKARD