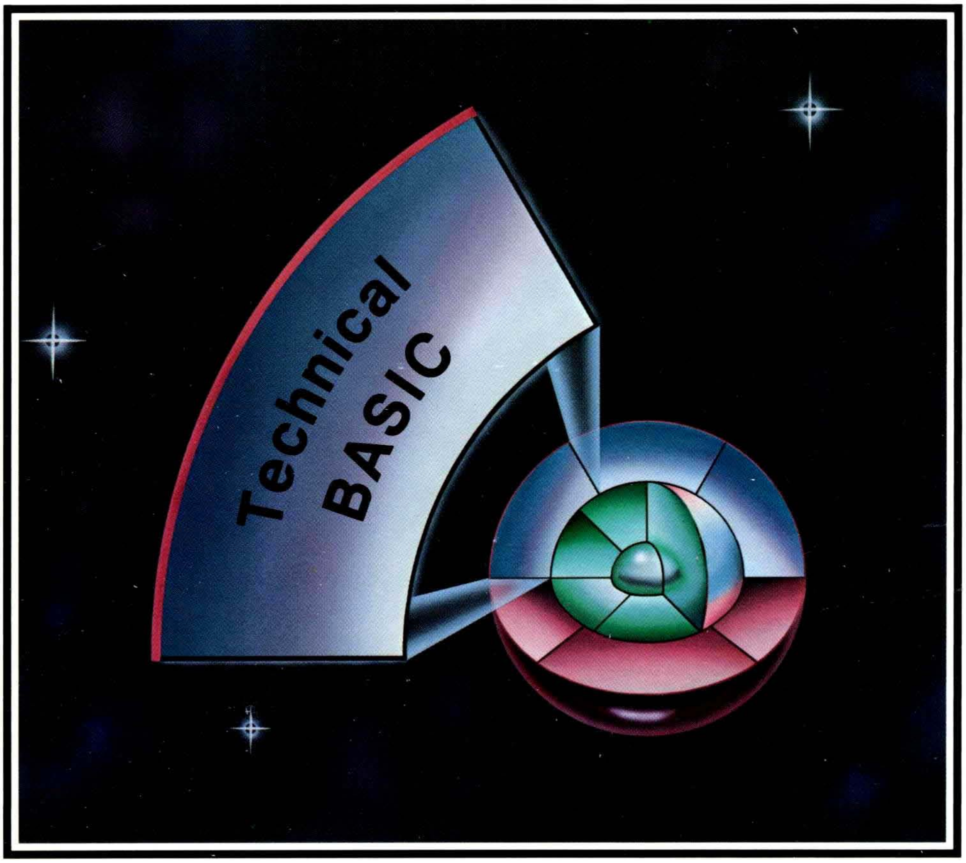
HP 9000 Computers

HEWLETT PACKARD

HP-UX Technical BASIC Programming Guide, Vol. 2



HP-UX Technical BASIC Programming Guide, Vol. 2

for HP 9000 Computers

HP Part Number 97068-90001

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing.date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

February 1986...Edition 1

Table of Contents

Chapter 9: Communicating with the Operator

Introduction 9–1
  Chapter Contents 9–1
Overview 9–2
  A Simplified Model 9–2
  Importance of the Human Interface 9–3
  General Design Suggestions 9–3
Sending Messages to the Operator 9–4
  Sending Audio Messages 9–4
  Displaying Messages on the Alpha Screen 9–5
  Printers 9–13
  Formatting Information 9–15
Accepting Messages from the Operator 9–21
  Types of Keyboard Inputs 9–21
  Softkeys 9–22
  Alphanumeric Input Methods 9–24
  Enabling and Disabling Keys 9–28
  Low-Level Keyboard Input Routines 9–29
  Reading Text from the Screen 9–30

Chapter 10: Using the Clock and Timers

Introduction 10–1
  Chapter Contents 10–1
Using the Clock 10–2
  Reading the Date 10–2
  Reading the Time of Day 10–2
  Time and Date Format Conversions 10–3
  Timing the Interval Between Events 10–5
Using the Timers 10–6
  Timer Interrupts 10–6
  Timer Functions 10–9
  Timers and Subprograms 10–10

Chapter 11: Data Storage and Retrieval
Introduction 11–1
Chapter Contents 11–2
Storing Data in Programs 11–3
Storing Data in Variables 11–3
Data Input by the User 11–4
Using DATA and READ Statements 11–4
Using BASIC/DATA Files 11–9
Brief Mass Storage Tutorial 11–9
Introduction to File Access Techniques 11–11
A Closer Look at BASIC/DATA File Access 11–14
A Closer Look at Serial Access 11–20
Random File Access 11–24
Determining Data Types 11–28
Trapping EOF and EOR Conditions 11–29
Using text/data Files 11–31
Writing to a text/data File 11–31
Reading from a text/data File 11–32

Chapter 12: C Binaries
Introduction 12–1
Chapter Contents 12–1
Overview 12–2
Structure of C Programs and Binaries 12–2
Compiling and Linking 12–3
Summary of Parameter-Type Matching 12–5
Restrictions 12–5
A Closer Look at Compiling and Linking 12–7
Example C Binaries 12–9
Passing Simple Numeric Parameters 12–9
Passing Numeric Array Parameters 12–11
Passing Simple String Parameters 12–13
BASIC String Length Headers 12–15
Passing String Arrays 12–17
Examples of C File I/O 12–19

Chapter 13: Pascal Binaries

Introduction 13–1
  Chapter Contents 13–1
Overview 13–2
  Structure of Pascal Programs and Binaries 13–2
  Compiling and Linking 13–3
  Summary of Parameter-Type Matching 13–5
  Restrictions 13–6
A Closer Look at Compiling and Linking 13–8
Example Pascal Binaries 13–10
  Passing Simple Numeric Parameters 13–10
  Passing Numeric Array Parameters 13–13
  Passing Simple String Parameters 13–17
  BASIC String Length Headers 13–19
  Passing String Arrays 13–21
  Using Files with Pascal Binaries 13–24

Chapter 14: FORTRAN Binaries

Introduction 14–1
  Chapter Contents 14–1
Overview 14–2
  Structure of FORTRAN Programs and Binaries 14–2
  Compiling and Linking 14–3
  Summary of Parameter-Type Matching 14–5
  Restrictions 14–6
A Closer Look at Compiling and Linking 14–8
Examples of FORTRAN Binaries 14–10
  Passing Simple Numeric Parameters 14–10
  Passing Numeric Array Parameters 14–12
  Passing Simple String Parameters 14–15
  Passing String Arrays 14–20
  Using Files with FORTRAN Binaries 14–22

Chapter 15: Graphics

  Chapter Contents 15–1
Example Graphics Programs 15–2
  Example of Graphics Output (Plotting) 15–2
  Example of Graphics Input 15–4
Determining Your Device's Capabilities 15–6
  List of Output (Plotting) Capabilities 15–6

Specifying Graphics Devices 15-9
    Explicitly Specifying a Plotting Device 15-9
    Explicitly Specifying Separate Input and Output Devices 15-11
Limits and Coordinates 15-12
    Overview 15-12
    A Closer Look at Coordinate Systems 15-15
    Axes and Grids 15-19
    User Units 15-22
    Changing Units: SETGU and SETUU 15-29
    Moving the Graphics Limits 15-32
    Aspect Ratio 15-34
    Plotting Boundaries 15-36
    Reflecting Images 15-48
    Digitizing Graphics Limits and Plotting Bounds 15-49
Graphics Input 15-52
    Input Capabilities 15-52
    What Is Digitizing? 15-52
    Digitizing the Input Locator's Position 15-52
Pens and Background 15-54
Moving the Pen and Drawing 15-56
    Moving and Drawing 15-57
    Line Types 15-64
Labeling 15-65
Plotting and Reading Pixels 15-69
    Plotting Pixels (BPLOT) 15-69
    Reading Pixels (BREAD) 15-70
    A Closer Look at Byte Plotting 15-70
    A Closer Look at Byte Reading 15-71
Storing and Retrieving Raster Images 15-73
    Storing Graphics in a File 15-73
    Retrieving Graphics from a File 15-73
    Dumping Graphics to Printers 15-74
Using Plotters 15-75
    Graphics Defaults Restored 15-75
    Additional Considerations 15-75
    Digitizing Plotter Pen Locations 15-76
    Graphics Using HPGL Commands 15-82

Index

Communicating with the Operator 9

Introduction

Have you ever been confused by the question posed by a program? Have you ever wondered which button to press next? Have you ever gotten a cryptic error message, or lost some important or irreproducible data? If you answered "yes" to any of these questions, then you know some of the frustrations of using a poorly designed computer/human interface[1].

As a programmer, you are on the other side of this interface. You have the responsibility of designing a program that others can use and, more importantly, will want to use. How will you ask questions? What assumptions are you going to make? How much time will you spend making your program easy to use? The time and effort you invest could mean the difference between a popular piece of software and one that everyone avoids like the plague.

Chapter Contents

This chapter describes the system features available for communicating with the computer operator. It contains these task and topics:

Task/Topics	Page
An overview of the elements in a human interface	9-2
General suggestions for improving computer/human communications	9-3
Sending audio messages	9-4
Sending alphanumeric messages	9-5
Accepting input from the softkeys	9-22
Accepting information from the keyboard	9-24

[1] A computer/human interface, or simply human interface, is informally defined to be the means by which the computer operator interacts with the computer. This interface includes hardware, software, and information.

Overview

In order to design an effective human interface for a program, you need to take a closer look at the operator/program communication process.

A Simplified Model

Although the human interface involves many aspects of the flow of information between computer to operator, here is a simplistic model of the communication process at this interface:

1. The **computer prompts** the operator for information.
2. The **operator receives** the message.
3. The **operator thinks** about the question, and then **formulates a response**.
4. The **operator makes the response**.
5. The **computer accepts the information** entered by the operator.

Communications Devices

These steps in the communication process are generally carried out by the following physical devices:

- **Computer output devices:** alphanumeric and graphics displays, beepers and voice-synthesis devices
- **Human input devices:** eyes, ears, sense of touch
- **Human information processor:** brain
- **Human output devices:** fingers and hands, voice
- **Computer input devices:** keyboards, graphics input devices (such as a knob, mouse, touch screen, and digitizer stylus), and voice-recognition devices

Other Factors

Along with these output and input devices, there are some other factors that affect the communication of information.

- Method of presenting the information (terminology, page layout, etc.)
- Placement of the preceding output and input devices
- Operator's past experience and present mental state
- Various other human factors

Importance of the Human Interface

In general, the most important function of a computer is to manipulate data. Although the computer can receive data from other computers and devices, it is probably more common that it gets data from a computer operator.

If you are the only person that uses a program you've written, then that program may not need a quality human interface. This normal requirement is eliminated because you know exactly what data the program needs, when it is needed, and how to enter it into the computer. However, if a program is used by other people, then the demands for a good human interface rise greatly — especially if they have different backgrounds. When the intended users do not understand computers, your program must be very skillfully written so that it does not confuse or intimidate the operator or make great demands on their computer expertise.

This part of the process of using software is one of the most error-prone, because it involves the subtly complex process of human communications. And the problem is further compounded because the humans are separated by space and time, as well as restricted to communicating with limited means — usually only visual computer prompts and manual human input.

General Design Suggestions

Good human interfaces don't just happen; they require effort, logical thinking, and thorough testing. In many programs, at least 25% of the code is dedicated to the human interface. And it is not unusual to use 60% of a good program for explanatory messages, operator interaction, error trapping, and so forth. Obviously, these estimates depend upon many factors, such as the task being performed and the intended operators; however, they do show that a significant portion of the program design effort should be devoted to the human interface.

Here is a brief list of general suggestions for developing an effective human interface.

- Ask simple, definite questions or prompts
- Present the questions and prompts in a natural, logical order
- Limit the set of alternative answers, if possible
- Supply a default answer, if possible
- Provide a chance for the operator to verify choice(s) picked or information entered
- Trap invalid operator responses, and give them another chance

Sending Messages to the Operator

The information you can send to the operator generally fits into these two categories:

- Descriptions of what the program is currently doing (or what mode it is currently operating in)
- Descriptions of what the user is expected to do

These "status reports" and "prompts" for information, respectively, may be made in one of these ways:

- With words (text)
- With pictures (graphics)
- With sound (auditory messages)

Let's deal with sound first, because of the simplicity of the related Technical BASIC features, and then with text. Graphics are covered in the "Graphics" chapter.

Sending Audio Messages

It would be a real attention getter to have the computer synthesize a pleasant voice that says: "Please don't touch the keyboard right now." However, sometimes a simple warning "beep" is enough to give the same message.

Generic Beeps

With some terminals and consoles[1], the only audio message available is the "bell" sound. This BASIC statement directs the terminal to make the bell sound.

```
BEEP
```

You can also get the same response by "displaying" the `bell` control character:

```
DISP CHR$(7)
```

This method works well when the operator probably knows what he is doing is wrong, but just needs a gentle reminder.

[1] The tones that your terminal or console can generate are listed in the *Implementation Specifics* appendix to the *HP-UX Technical BASIC Language Reference*.

Varying Tones

On other terminals and consoles there is a tone generator, which you can use to produce sounds of varying frequency and duration. Execute this statement to see if your machine has these capabilities.

```
BEEP 10,10 @ BEEP 20,20
```

If you get two different pitches, then your hardware has these capabilities.

The first parameter in the BEEP statement controls the *frequency*, while the second controls the *duration*.

```
BEEP Frequency,Duration
```

The range of the frequency and duration parameters are given in the *Specifics* appendix for your system.

Displaying Messages on the Alpha Screen

The mechanics of using alpha displays with HP-UX Technical BASIC systems vary from system to system. This section contains some general information about using displays with all systems. Consequently, you may want to refer to the *Getting Started Guide* for your Technical BASIC system as you read this section.

Using printers to display information is described in a subsequent section.

The Essence of Displaying Messages

Giving instructions to the operator can be condensed into these basic steps:

1. Clear the display of any irrelevant information.
2. Make sure that the display device is operating in the proper mode (for instance, not in insert mode)
3. Use as much of the display as necessary to give unambiguous, understandable instructions.

In the early days of computers, memory was a scarce and expensive resource. Programmers were encouraged to use as little memory as possible. It seemed as though there was a contest to see who could put the most information into a short message.

Please realize that those days are over. Take a typical HP-UX system as an example. The standard machine is shipped with over a half-million characters of memory, and there is no significant restriction on program size. Neither is there any real restriction due to the display size, since most HP displays supported on Technical BASIC usually have at least 20 lines of 80 characters each visible at all times. It is generally false economy to display tiny, cryptic messages.

Which Device Is the Display Screen?

Statements that display text (like DISP and CAT) send the characters to the current CRT IS device. Normally this device will be the screen on which characters appear as you type[1].

You can see which screen is the CRT IS device by executing the following statement:

```
DISP "This is the current CRT IS device."
```

The display (or printer) on which the message appears is currently the CRT IS device. Normally it will be your terminal or console screen. However, if these characters are currently being sent to a printer (or file), then you can specify that your console is now to be the display device by executing this statement:

```
CRT IS 1
```

The numeric parameter 1 specifies the screen's *device selector*.

You can also specify that a file is to be the CRT IS device. If a file does not exist, then you can create one for this purpose. Then assign a *file selector* to the file, and specify this file number as the CRT IS device.

```
CREATE "CRTISFile",1
ASSIGN 11 TO "CRTISFile"
CRT IS 11
```

The CREATE statement creates a text/data[2] file in the current working directory. The ASSIGN statement assigns a file selector of 11 to the file named CRTISFile. Since no directory path was specified, the file was assumed to be in the current working directory. If there is no file named CRTISFile in that directory, then the system automatically creates it.

Subsequent information that would normally be sent to the screen (such as output of CAT, LIST) will be sent to this file. If the file already exists and has information in it, then the subsequent information is appended to the file.

You can also specify a **screenwidth** in the CRT IS statement. For instance:

```
CRT IS 1,65
```

[1] If you see no characters on your screen as you type at the keyboard, press the carriage-return key, then type in an **ALPHA** command, and execute the command by pressing the carriage-return key again. Executing this statement turns on the alpha display.

[2] You can read files of this type from BASIC by using **ASSIGN** to open the file and then reading lines of text with the **ENTER**s statement. For example, see "Using text/data Files" in the "Data Storage Retrieval" chapter.

BASIC will subsequently allow the `DISP` statement to display only 65 columns of text on the screen. (Other methods of writing to the screen are not affected, however.)

Now that you have seen how to determine which display you will be using, and how to specify another, the next step is to find out what you can do with it.

Determining Display Capabilities

An inherent requirement of using the steps above is knowing (or determining) the display device's capabilities. If you don't know, for instance, the width of an alphanumeric display screen, then you might try to put more than one line of text on a display line.

Here are some relevant questions you might ask about a display device's attributes and capabilities:

- What is the screen's width (number of columns) and height (number of rows)?
- What characters can it display, including enhancements (such as half-bright and underlining)?
- Can you position its cursor? (The cursor is a pointer that indicates the location at which the next character will be displayed.)
- Does it have special insert or delete modes or operations?
- Can you scroll the text on the display?

There are several **approaches** that you can take to determine a display device's capabilities:

- Read the display device's documentation.
- Observe its operation.
- Have the program determine them.

Using the **first approach**, you can read a display device's documentation, which is usually shipped with the device. For instance, if you are using an Integral Personal Computer, then you can read its installation and operating manuals. A list of the characters it can display, along with operating modes and escape code sequences it implements, is provided in the *Implementation Specifics* appendix shipped with the Integral HP-UX Technical BASIC system.

Multi-user systems, such as Series 500 HP-UX, are capable of supporting several different terminals at one time. In this case, you will need to read the documentation for each terminal to determine its capabilities.

Using the **second approach**, you could begin displaying some character codes on the screen and observe the results. You should eventually do this anyway to get a feel for what is pleasing to the eye and effectively conveys the desired information.

The following example shows an application of the **third approach**: determining display width with a BASIC program[1].

```
100  DIM Lines$[170]
110  Lines$=RPT$(" ",40)&"40"&RPT$(" ",8)&"50" ! String pos 41,51.
120  Lines$=Lines$&RPT$(" ",28)&"80" !          String pos 81.
130  Lines$=Lines$&RPT$(" ",78)&"160" !         String pos 161.
140  !
150  CLEAR !        Clear the screen.
160  ALPHA 1,1 !    Home the cursor (Row 1, Column 1).
170  AWRIT Lines$ ! Write line (excess will "wrap" to next line).
180  !
190  ALPHA 3,1 !    Move cursor to start of second line.
200  AREAD Lines$ !  Read characters (which will show width).
210  ScreenWidth=VAL(Lines$) !  Convert string to number.
220  ALPHA 6,1 !     Move cursor to start of sixth line.
230  DISP "Width of screen=";ScreenWidth;"characters."
240  !
250  END
```

Here are the results of running the program on an 80-column screen.

```
                                        40        50

80

160
Width of screen= 80 characters.
```

[1] The **AREAD** statement works on "line-oriented" terminals. A "line-oriented" terminal is one that can send and receive characters a line at a time. If you can type in a BASIC statement or command, execute it, move the cursor back onto the line, and then *successfully* re-execute it, you have a "line-oriented" terminal.

The program creates a string (Lines$) that is longer than any line that any screen can display. It places characters such as "40", "50", and so forth at string locations 41, 51, and so forth, respectively. (Note that there is a space between 80 and 160 in the above results. This is the because a line-feed is inserted after the eightieth character on a line.) The ALPHA statement next positions the cursor at column 1 of row 1. The AWRIT statement writes this string into screen memory beginning at the current cursor location. Since the length of the string is greater than a screen width, some of the characters will be placed ("wrapped") onto subsequent row(s) of the screen. The AREAD statement then reads the number on the third row, which represents the width of the screen. The VAL function converts the string read by AREAD into a numeric value, which is assigned to the numeric variable named ScreenWidth. You can use an analogous technique to determine the number of lines (rows) on the screen.

Although this is a way for the program to determine the screen's width, it may not be the most reassuring thing for a program's user to see as he begins using the program.

An **alternate method** of programmatically determining screen width might be as follows: set up a table that lists each type of terminal's capabilities; have the program ask the user to identify the product number of the screen device; access the entry in the table that describes that device's capabilities; set up the communication model for that terminal based on the device's capabilities.

Clearing the Screen

It is confusing to the operator (and embarassing to the programmer) when two or more displays combine in an unplanned manner. The culprits are often "left-over" alpha and graphics.

To completely erase the alpha display, use this statement:

```
CLEAR
```

It moves the cursor to its "home" position (upper, left corner), scrolling the text if necessary, and clears all characters from the display.

To completely clear left-over graphics, execute GCLEAR. Note that alpha and graphics are displayed separately on some consoles and terminals, but are displayed simultaneously on others.

Turning Off Unwanted Modes

As another example, suppose that the previous user left the cursor in the middle of a screen of text with "insert mode" left on. If a subsequent program attempts to display new text without turning off insert mode and clearing the screen, then the result may be a chaotic screen.

The DISP statement does not provide a high-level method for getting the display out of modes like "insert character." Those modes are controlled by sending an escape sequence to the display. In this case, you will need to cancel the insert mode (return to not inserting characters before current cursor position). Here is a simple example:

```
440  CancelInsert$=CHR$(27)&"R"
450  DISP CancelInsert$;
```

The PRINTALL statement directs the system to print all information that is sent to the display screen; the information is printed on the current PRINTER IS device. You can cancel this mode by executing a NORMAL statement.

Positioning the Cursor

Whenever you execute a statement that displays characters on the screen, these characters are displayed beginning at the current *cursor location*. For instance, one of the preceding examples showed a method of moving the cursor. Here is a similar example:

```
100  DIM Chars$[170]
110  Chars$="Cursor location."
120  !
130  CLEAR !        Clear screen, and "home" cursor (row 1, column 1).
140  !
140  DISP Chars$ !  Display beginning at cursor location.
150  DISP Chars$
170  !
180  ALPHA 3,20 !   Move cursor to line 3, column 20.
190  AWRIT "Cursor location doesn't change."
200  AWRIT "With AWRIT, loc"
210  !
220  END
```

Here are the program's results:

```
Cursor location.
Cursor location.
                  With AWRIT, loc doesn't change.
```

The CLEAR statement clears the display and sets the cursor location to row 1 and column 1. The subsequent DISP statement displays characters beginning at this location. As the DISP statement finishes, it automatically moves the cursor to the next line by sending an "end-of-line" (EOL) sequence: a carriage-return control character followed by a line-feed control character.

The cursor is then moved to column 20 of row 3 with the ALPHA statement. The AWRIT statement then writes the specified characters on the display. AWRIT is different from DISP in that it does **not** update the cursor location, as shown by the second AWRIT statement beginning at the same location (3,20) and overwriting characters written by the first one.

Determining the Cursor's Location

If you are not sure where the cursor is, you can determine its location by using the CURSROW and CURSCOL functions.

- CURSROW returns the row.
- CURSCOL returns the column.

You can use these functions just as you would other numeric system functions. Here is an example of using them in a program.

```
100  Star$="*"
110  !
120  CLEAR
130  FOR RowNumber=1 TO 16 STEP 3
140    Col_=RND *60 !              Random column.
150    ALPHA RowNumber,Col_ !      Move cursor.
160    AWRIT Star$ !               Display the "*".
170    Row_=CURSROW !              Determine row.
180    Col_=CURSCOL !              Determine column.
190    ALPHA ,CURSCOL +3 !         Move cursor (relative).
200    DISP "(";Row_;",";Col_;")" ! Show row and column.
210  NEXT RowNumber
220    !
230  END
```

Here are typical results of running the program:

```
                                            * ( 1 , 43 )

                                  * ( 4 , 33 )

                                      * ( 7 , 37 )

                                     * ( 10 , 36 )

              * ( 13 , 14 )

         * ( 16 , 7 )
```

Turning the Cursor On and Off

The cursor is the screen location at which subsequently typed or displayed characters will begin appearing. Normally the cursor's location is indicated by an inverse-video block or a blinking underline character.

To disable the visual cursor, execute this statement[1]:

```
OFF CURSOR
```

To re-enable the visual cursor, execute:

```
ON CURSOR
```

Displaying Blank Lines

If the cursor position is at the start of a blank line when `DISP` is executed, that line remains blank. However, if there is text on that line, the text remains. This behavior is due to the fact that a `DISP` statement with no parameters simply sends an end-of-line sequence, which is a different operation than writing a line of blank characters — ASCII spaces, or `CHR$(32)`. This is not to say that it is "wrong" to use `DISP` with no parameters. It just means that you cannot guarantee the output of a blank line by using `DISP` with no parameters.

To print a blank line, blanks must be printed. One of the most convenient ways to send a line full of blanks is to use the `TAB` function. Here is a sequence that prints three blank lines:

```
100  ScreenWidth=80 !  This may vary for your display device.
110  DISP TAB(ScreenWidth)
120  DISP TAB(ScreenWidth)
130  DISP TAB(ScreenWidth)
```

Before getting into greater detail about formatting information that you sent to the display, let's take a look at what capabilities you have for sending information to printers.

[1] This feature is not implemented on some consoles and terminals.

Printers

The mechanics of using printers with HP-UX Technical BASIC systems vary from system to system. This section contains some general information about using printers with all systems. Consequently, you may want to refer to the *Getting Started Guide* for your Technical BASIC system as you read this section.

The PRINT statement sends information to a printer in the same fashion as the DISP statement sends information to a screen display. The device specified in the last PRINTER IS statement, or the default system printer[1], receives PRINT statements' output[2].

To see which printer is the current PRINTER IS device, execute this statement:

```
PRINT "This is sent to the PRINTER IS device."
```

You can also specify that another device is to be the system printer. Here is an example of creating a file and then specifying that the file is to be the system printer.

```
CREATE "PRTISFile",1
ASSIGN 11 TO "PRTISFile"
PRINTER IS 11
```

The CREATE statement creates a text/data file in the current working directory. The ASSIGN statement assigns a *file selector* of 11 to the file named PRTISFile. Since no directory path was specified, the file was assumed to be in the current working directory. If there is no file named PRTISFile in that directory, then the system automatically creates it.

There are times when you want to have printed records of what has been displayed on the screen. The PRINT ALL statement directs the system to print a copy of whatever information is sent to the display on the current PRINTER IS device.

If you have not been operating in PRINT ALL mode, but you find that you need to get a printed version of what is currently on the screen, you can use the DUMP ALPHA[3] statement to send a copy to the current PRINTER IS device.

1 With single-user Integral HP-UX systems, the default system printer is the built-in printer. With other single-user and multi-user HP-UX systems, the default system printer is the display screen.

2 Multi-user HP-UX systems use intermediate files to receive the output of PRINT statements, which are then "spooled" to the printer.

3 DUMP ALPHA requires that the printer is capable of displaying graphics. Note also that it is not implemented on some terminals. For further details, see the "Graphics" chapter of the *HP-UX Technical BASIC Programming Guide* (this manual) for your particular HP-UX Technical BASIC system.

A Typical Printer's Character Set

Most ASCII characters are printed on an external printer much like they appear on the display screen[1]. However, depending on your printer, there will be exceptions. Several printers will also support an alternate character set; this alternate set is often a foriegn 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. This section describes typical characters that printers can print and use as control information.

Control Characters

In addition to a "printable" character set, printers usually respond to control characters. These non-printing characters produce a response from the printer. The following table shows some of the control characters and their effect.

Table 9-1. Control Characters

Printer's Response	Control Character's ASCII Value	Keyboard Character
ring printer's bell	7	CTRL-G
backspace one character	8	CTRL-H
horizontal tab	9	CTRL-I
line-feed	10	CTRL-J
form-feed	12	CTRL-L
carriage-return	13	CTRL-M

One way to send control characters to the printer is with the CHR$ function. Execute the following.

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

The printer usually responds by executing a form-feed — it moves the paper to the beginning of the next blank sheet, and re-positions the print head to the beginning of the first line.

Other control characters may be valid for your printer. For example, sending a control-N to the 82905B printer changes the character size of subsequent text.

```
30  Big$=CHR$(14)
40  PRINT Big$;"Double-Width Text"
50  END
```

[1] A list of the characters available on a particular printer is given in the documentation sent with that printer.

Refer to the appropriate printer manual for a complete listing of control characters and their effect on your printer. Note that some printers allow control characters to affect only the line of text on which they were used.

Escape-Code Sequences

Similar in use to control characters, escape-code sequences allow additional control over most printers. These sequences consist of the escape control character, `CHR$(27)`, followed by one or more characters.

For example, the 2631A printer is capable of printing characters in several different fonts. To print extended characters on this printer, an escape code sequence is sent to the printer before the actual text to be printed is sent.

```
20  Esc$=CHR$(27)
30  Big$="&k1S"
40  Regular$="&k0S"
50  PRINT Esc$;Big$;"Extended-Font Text"
60  PRINT Esc$;Regular$;"Back to normal."
70  END
```

Since each printer may respond differently to control characters and escape code sequences, check the manual that came with your printer.

Formatting Information

This section describes how to use the `DISP` and `PRINT` statements to "format" the information you print.

For many applications the `PRINT` or `DISP` statement provides adequate formatting. The simplest method of formatting is by specifying a comma or semicolon between items.

When the comma is used to separate items, the items are aligned on field boundries. Fields start in column one and occur every 21 columns (columns 1,22,43,64,...). Here is an example of this type of formatting with `PRINT` statements:

```
PRINT "1234567890123456789012345678901234567890";
PRINT "012345678901234567890123456789"
DATA 1.1,-22.2,300000,5.1E+8
READ A,B,C,D
PRINT A,B,C,D
```

Here are the results:

```
1234567890123456789012345678901234567890123456789012345678901234567890
 1.1                 -22.2                300000               510000000
```

Using the semicolon as the separator causes the numbers to be put as closely together as the compact form allows. The compact form always uses one leading space (or — when the number is negative) and one trailing space. That is why the positive numbers in the previous example appear to print one column to the right of the field boundries. The next example shows how the compact form prevents numeric values from running together.

```
PRINT "1234567890123456789012345678901234567890";
PRINT "0123456789012345678901234567890123456789"
DATA 1.1,-22.2,300000,5.1E+8
READ A,B,C,D
PRINT A;B;C;D
```

Here are the results:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
 1.1 -22.2  300000  510000000
```

The comma and semicolon are often all that is needed to format a simple table.

You can also format the entire contents of an array, using the comma or semicolon to control the format of the output. Here is an example of printing an array in which each array element contains the value of its subscript:

```
10 OPTION BASE 1
20 DIM A(5)
30 DATA 1,2,3,4,5
40 READ A(1),A(2),A(3),A(4),A(5)
50 PRINT A(1);A(2);A(3);A(4);A(5)
60 END
```

Here are the results:

```
 1 2 3 4 5
```

Another method of aligning items is to use the TAB function.

```
10  PRINT "1234567890123456789012345678901234567890"
20  PRINT TAB(16);"*"
30  END
```

Here are the results:

```
1234567890123456789012345678901234567890
               *
```

A more powerful formatting technique employs the ability of the PRINT and DISP statements to use an image to specify the format.

Using Images

Just as a mold is used for a casting, an image can be used to format data. An image specifies how each item should appear. The computer then attempts to format the items according to the image.

One way to specify an image is to include it in the PRINT statement. The **image specifier** is enclosed within quotes and consists of one or more **field specifiers**. A semicolon then separates the image from the list of items to be printed.

This statement prints the value of π (3.141592654...) rounded to three digits to the right of the decimal point.

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

Here is its result:

```
3.142
```

For each character "D" within the image, one digit is 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.14D";PI
```

```
3.14159265358979
```

Instead of typing fourteen "D" specifiers, one for each digit, a shorter notation was used to specify a repeat factor before the digit field specifier. The image "DDDDDD" is the same as the image "6D".

The image specifier can be included in the PRINT or DISP statement or on its own line. When the specifier is on a different line the PRINT or DISP statement accesses the image by either its line number or line label.

```
100   Format: IMAGE 6Z.DD,X
110   DATA 1.5,25.57,.056,-.555,-3.4,-88.9
120   READ A,B,C,D,E,F
130   PRINT USING Format;A,B,C
120   PRINT USING 100;D,E,F
150   END
```

Executing this program gives the following results:

```
000001.50 000025.57 000000.06
-00000.56 -00003.40 -00088.90
```

Notice that the image specifier Z filled the field to the left of the radix with zeros.

Numeric Image Specifiers

Several characters may be used within an image to specify the appearance of a numeric value.

Table 9-2. Numeric Image Specifiers

Image Specifier	Purpose
D	Replace this specifier with one digit of the number to be printed. If the digit is a leading zero, then print a space. If the value is negative, then one leading space may be used by the negative sign.
Z	Same as "D" except that leading zeros are printed.
E	Prints two digit of the exponent after printing the sequence "E+". This specifier is equal to "ESZZ". See the *Technical BASIC Language Reference* for more details.
K	Print the entire number without leading or trailing spaces.
S	Print the sign of the number: either a "+" or "-".
M	Print the sign if the number is negative; if positive, print a space.
.	Print the decimal point (radix).
R	Print the comma radix.

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 "ZZZ.3D";33.666             033.666

   PRINT USING "ZZZ";.444                  000
   PRINT USING "ZZZ";5.55                  005

   PRINT USING "SD.3DE";6.023E+23          +6.023E+23
   PRINT USING "S3D.3DE";6.023E+23         +602.300E+21
   PRINT USING "S5D.3DE";6.023E+23         +60230.000E+19
```

To specify multiple fields within the image, the field specifiers are separated by commas.

```
PRINT USING "K,5D,5D";100,200,300

100  200  300

PRINT USING "ZZ,DD,DD";1,2,3

01 2 3
```

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

```
10 PRINT "123456789012345678901234567890123"
20 PRINT USING "5D.DD" ; 3.98,5.95,27.5,129.95
30 END
```

This program produces the following after execution:

```
123456789012345678901234567890123
    3.98    5.95   27.50  129.95
```

The image is re-used for each value. However, an error will result if the number cannot be accurately printed by the image specifier. For instance, the number 20 cannot be accurately printed by the "D" image specifier, since it requires at least two significant digits.

String Image Specifiers

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

Table 9-3. String Image Specifiers

Image Specifier	Purpose
A	Print one character of the string. If all characters of the string have been printed, then print a trailing blank.
X	Print a space, `CHR$(32)`.
"literal"	Print the characters between the quotes.

Note that the long strings of numbers above the results are used to show column spacing they are not part of the result. The same type of long number strings were used in previous programs for the same purpose but they were part of the program output.

The following examples show various ways to use string specifiers.

Executing these statements:

```
PRINT "1234567890123456789O1234567"
PRINT USING "5X,10A,2X,10A";"Tom","Smith"
```

Produces the following results:

```
1234567890123456789O1234567
     Tom         Smith
```

Executing these statements:

```
10 IMAGE 5X,"John",2X,10A
20 PRINT "1234567890123456789012"
30 PRINT USING 10;"Smith"
40 END
```

Produces the following:

```
1234567890123456789012
     John  Smith
```

Executing these statements:

```
10 IMAGE "PART NUMBER",2X,10D
20 PRINT "12345678901234567890123"
30 PRINT USING 10;90001234
40 END
```

Produces the following:

```
123456789012345678901234
PART NUMBER    90001234
```

Additional Image Specifiers

The following image specifiers serve a special purpose.

Table 9-4. Additional Image Specifiers

Image Specifier	Purpose
B	Print the ASCII character whose code is given by the "binary" number. (This is similar to the `CHR$` function.)
#	Suppress the otherwise automatic end-of-line sequence (carriage-return and line-feed).
/	Send an end-of-line sequence.

Examples

To print a form-feed but suppress the automatic end-of-line sequence, execute the following:

```
PRINT USING "#,B";12
```

To print the ASCII characters that correspond to the codes given by three integers, execute the following statement:

```
PRINT USING "B,B,B";67,97,116
```

The following appears on the display:

```
Cat
```

Accepting Messages from the Operator

There are several ways to get data from the operator:

- From the keyboard
- From a positioning device (such as a mouse or graphics-tablet stylus)
- From an audio input device

The main focus of this section is on inputs from the keyboard. Inputs from positioning devices are described in the “Graphics” chapter. Audio input is beyond the scope of this book.

Types of Keyboard Inputs

There are two general methods of getting operator input through the keyboard:

- With softkeys
- With alphanumeric keys

Softkeys

When possible, using softkeys is a very good choice. It limits the number of alternative inputs, thereby eliminating the need for translating an endless variety of typing mistakes that might be made by the operator. Another benefit is that softkey input is very tightly controlled by the programmer. For information on softkeys as "typing-aids" read the section "Using Typing-Aids Keys" in the *Technical BASIC Getting Started Guide*.

`ON KEY#` statements are used to set up and enable interrupt service routines to be executed when each softkey is pressed[1]. The `KEY LABEL` statement updates the screen with visual reminders of each softkey's definition.

```
100  ! This program shows simple usage
110  ! of the softkeys and system clock.
120  !
130  CLEAR !     Clear alpha screen.
140  OFF CURSOR ! Disable visual cursor.
150  !
160  ! Set up softkey definitions.
170  ON KEY# 1,"Start" GOSUB Starts
180  ON KEY# 2,"Stop" GOSUB Stops
190  ON KEY# 3,"Reset" GOSUB Resets
200  ON KEY# 4,"Lap" GOSUB LapTime
210  KEY LABEL ! Show softkey labels on screen.
220  !
230  ! Set up initial screen.
240  ALPHA 2,1 @ DISP "Time of day:",TIME$
250  ALPHA 4,1 @ DISP "Start time:",HMS$(0)
260  ALPHA 6,1 @ DISP "Elapsed time:",HMS$(0)
270  ALPHA 8,1 @ DISP "Lap time:",HMS$(0)
280  !
290  Loop: ALPHA 2,22 @ DISP TIME$
300        WAIT 1000 !   Dummy delay.
310        IF NOT Timing THEN Loop !  Don't update elapsed.
320        ALPHA 6,22 @ DISP HMS$(TIME-Tstart) ! Elapsed.
330      GOTO Loop
340      !
350 Starts: Tstart=TIME
360          Timing=1 !    Set flag.
370          ALPHA 4,22 @ DISP HMS$(Tstart)
380        RETURN
390         !
400 Stops: Timing=0 !     Clear flag.
410      RETURN
420      !
```

[1] Service routines are described in the "Program Structure and Flow" chapter.

```
430 Resets: Timing=0 !    Clear flag.
440          ALPHA 4,22 @ DISP HMS$(0)
450          ALPHA 6,22 @ DISP HMS$(0)
460          ALPHA 8,22 @ DISP HMS$(0)
470         RETURN
480         !
490 LapTime: IF NOT Timing THEN RETURN
500           ALPHA 8,22 @ DISP HMS$(TIME-Tstart)
510         RETURN
520         !
530         !
540  END
```

Here is a typical starting screen produced by the program:

```
Time of day:          14:35:45

Start time:           00:00:00

Elapsed time:         00:00:00

Lap time:             00:00:00
```

When the program begins execution, only the time of day is updated. The other times shown remain 00:00:00.

Pressing the [Start] key initiates a branch to the subroutine named flag" by assigning a value of 1 to the variable named Timing. When this flag is set, the elapsed time is shown along with the time of day when the timing was started (in the "Loop" segment).

Pressing the [Lap] key initiates a program branch to the LapTime subroutine, which displays the time elapsed since the [Start] key was pressed.

Pressing the [Stop] key initiates a branch to the Stops subroutine, which halts timing by clearing the timing flag. When the main loop is executed subsequently, the elapsed time is no longer updated.

Pressing the [Reset] key initiates a branch to the subroutine named Resets. This routine displays 00:00:00 for Start, Elapsed, and Lap times.

The program employs several techniques of moving the cursor and displaying data that were shown earlier in this chapter.

Alphanumeric Input Methods

Unfortunately, it is often necessary to leave the comfortable, controlled world of softkeys. For instance, suppose you need to get a number, such as a device selector, from the operator. Valid values of device selectors range from 1 through 1030. You can't very well define a softkey that increments a variable and expect the operator to press it several hundred times! Instead, you will normally ask the operator to use numeric keys to enter the number.

There are two methods that you can use to accept alphanumeric inputs from the keyboard:

- Use `INPUT` or `LINPUT` to enter values that can be assigned to string and numeric variables.
- Use `ON KYBD` to input individual keystrokes.

With the `INPUT` and `LINPUT` statements, the operator can type in information, use the cursor control or backspace keys to edit the data if necessary[1], and then press the carriage-return key to send the data to the BASIC system for evaluation and assignment to corresponding BASIC variable(s). This method is the "high-level" approach to accepting keyboard input, since it lets the system handle the often tricky details of moving the cursor, displaying and erasing characters on the screen, and so forth.

With the `ON KYBD` statement, each key is handled individually by a service routine, and you, the programmer, have to implement any desired editing capabilities. The `ON KYBD` method is the "low-level" method, since it involves much more detail; however, it gives the program greater control and flexibility.

With both of these methods, you can use the `ENABLE KBD` statement to enable or disable certain keys (or groups of keys). For instance, you can disable the [Reset] and [Break] keys[1] while still allowing the typing-aid and alphanumeric keys to function normally.

Before getting into the details of using these methods, here are some general suggestions that apply to all methods of accepting keyboard inputs.

[1] These keys may be labeled differently on your particular keyboard. See the subsequent section called "Enabling and Disabling Keys" for further details.

Anticipate Common Problems

One task that can be performed by the input routine is to anticipate common problems. Many techniques are covered in this section's examples, but here is a preview.

- You know that exceeding the dimensioned length of a string gives error 18, so don't use short string variables in an `INPUT` statement.
- You know that `CAPS LOCK` might be on or off when the operator starts typing, so use the `UPC$` string function to convert the inputs to uppercase characters before comparing them to string constants.
- You know that an operator is likely to just execute `CONT` (continue) if he isn't sure how to respond, so make sure that your input routine can handle a null response and that it assigns a reasonable default answer for such inputs.

Error Trapping Simplifies Input Routines

No matter how much time you have spent anticipating possible errors and making an input routine "bomb-proof," you can always find someone who can enter an incorrect response. However, don't feel bad, because the proper handling of keyboard input may be one of the most difficult areas of applications programs. Instead of writing elaborate input routines that can parse broken English with misspelled words, you can use the `ON ERROR` mechanism to trap errors that have not been (or cannot be) anticipated. The objective in such an approach is two-fold: to keep the program running, and to give the operator a chance to correct the mistake.

Here is a typical example. You ask the operator for a file name. Your program can't tell if the operator entered the name of a file that exists until it accesses the disc. The `ON ERROR` routine can tell the operator that the file does not exist on the specified (or default) volume and then ask for another file name. See the "Handling Errors" chapter for more information on error-trapping techniques.

The Two High-Level Input Methods

As mentioned before, there are two keywords available for accepting alphanumeric keyboard inputs:

- `INPUT`
- `LINPUT`

Both statements allow you to enter string values into BASIC variables; however, only `INPUT` also allows inputs into numeric variables. Here is an examples of using `INPUT`:

```
100  ON ERROR GOTO AskEmplNum  ! Set up error trap.
110  !
200 AskEmplNum: DISP "Please enter your employee number."
210             INPUT EmplNum
220             DISP "Is this correct ? (Y/N)",EmplNum
230             INPUT Answer$
240  IF UPC$(Answer$)<>"Y" THEN AskEmplNum
250  OFF ERROR !               De-activate error trap.
```

The example first sets up an `ON ERROR` branch to the beginning of the input routine. Let's look at how the routine would be executed *without* input errors before describing its error-trapping behavior.

The program displays a prompt for information (with `DISP`), and then directs the system to await numeric input data (with `INPUT`). The operator is then expected to type in a number and press the carriage-return key (to send the data to the system for evaluation and assignment into the numeric variable named EmplNum). If the operator enters a "valid" number, then program execution continues with the next line (220).

Next the program "echoes" the input data on the screen and asks the operator to verify that it is correct. If it is, then the operator is expected to type a "Y" and press the carriage-return key again. Note that this section anticipates a common problem — lettercase disagreement — by converting that the first character of the answer to an uppercase letter before comparing it with the uppercase "Y" that indicates an affirmative response.

Now back to the error-trapping mechanism. This is probably the simplest form of error trapping during input from the operator; it merely asks for the operator to input data again. The program will continue to do so until there are no run-time errors during the program. A typical error would be the operator entering string data with no numeric characters. In such case, the system would normally report: `Error 43 on line 210 NUMERIC INPUT REQUIRED`. However, since this program branches to AskEmplNum upon detecting an error, the error report is disabled and the operator is asked again to enter the number.

Here is a similar example that uses LINPUT. (LINPUT stands for "Literal INPUT".)

```
100  ON ERROR GOTO AskIncome    ! Set up error trap.
110  !
120 AskIncome: LINPUT "Monthly income?",Income$
130            LINPUT "Is this correct ? (Y/N)  "&Income$, Answer$
140  IF UPC$(Answer$)<>"Y" THEN AskIncome
150  OFF ERROR !                 Disable error trap.
```

This program uses LINPUT for the primary reason that most people (in America, anyway) use commas in numbers between the hundreds and thousands places, and so forth. For example: 1,500.00. If you tried to use INPUT to enter this number into a numeric *or string* variable, you would get an erroneous value of 1. This is because INPUT interprets the comma as a field separator. Using the LINPUT statement allows you to enter the number, commas and all, into a string variable. The program can then parse the string to remove the commas, if necessary.

The preceding examples show that there are several differences between INPUT and LINPUT.

The **main advantages of INPUT** are as follows:

- Either numeric or string values can be input.
- A single INPUT statement can process multiple fields, and those fields can be a mix of string and numeric data.

The INPUT statement can be powerful and flexible. When you know the skill level of the person running the program, INPUT can save some programming effort. However, this statement does demand that the operator enter the requested fields properly.

Two of the **disadvantages of INPUT** are as follows:

- Improper entries to numeric variables can cause errors, such as Error 130 NUMERIC VALUE REQUIRED and Error 2 OVERFLOW.
- Certain characters can cause problems. Commas and quote marks have special meanings and are the primary offenders.

The problem with INPUT is that the program is powerless to overcome the disadvantages. If you are asking for a numeric quantity and the operator keeps trying to enter a name, the program will never leave the INPUT statement. The BASIC system will display error 43 until the operator either gets tired or realizes the mistake. In the event of an error, the computer automatically re-executes the INPUT statement until the operator satisfies all the requirements. Your program never gets a look at the input, because the erroneous input initiates a branch back to the beginning of the input routine.

The LINPUT statement can help with these potential problems. The result of any LINPUT statement is a single string that contains an *exact image* of what the operator typed. If no data are input, then the variable is given the value of the "null string" (a string of length 0 characters). If you need things like default values, numeric quantities, and multiple values, then you will need to process the string after you get it.

Since LINPUT accepts any characters without any special considerations, the only normal error would be string overflow. If the string used to hold the LINPUT characters is dimensioned to hold a line of text (usually 80 characters) or more, then it becomes highly unlikely that the operator will overflow the string from the keyboard. Therefore, LINPUT is a very "safe" way to get data from the keyboard.

To find out further details regarding the use of INPUT and LINPUT, see the *HP-UX Technical BASIC Language Reference*.

Enabling and Disabling Keys

You can use the ENABLE KBD statement to enable and disable certain keys and groups of keys. For instance, you can disable the special function keys (during program execution) by executing this statement:

```
100  ENABLE KBD 255-2^5 ! All bits 1, except bit 5.
```

The numeric parameter is the *mask* that specifies which keys are to be enabled or disabled; a 1 in a certain bit position enables that key (or group), while a 0 disables the key(s). Bit definitions in the mask are shown in the *HP-UX Technical BASIC Language Reference*.

Here are the two keys and two key groups that can be enabled and disabled with this statement:

- [Reset]
- [PAUSE][1]
- Special function keys (or "softkeys")
- All other keys (such as alphanumeric and cursor-control keys)

The corresponding keycap labels are shown in the *Getting Started* manual for your particular Technical BASIC system.

[1] Information for pausing a program is found in the *HP-UX Technical BASIC Getting Started Guide* in the chapter "Running Programs" under the section entitled "Pausing and Continuing a Program."

Low-Level Keyboard Input Routines

With Technical BASIC, you have the capability of trapping every keystroke using the ON KYBD statement. You can use this feature to design very effective keyboard interfaces. However, the programming effort for this type of application is often relatively large. In fact, using ON KYBD to accept keyboard input while displaying a cursor and positioning text is essentially writing a text editor. Unfortunately, programs of that magnitude are beyond the scope of this manual.

Here is an example that shows a simple usage of ON KYBD to detect presses of alphanumeric keys.

```
100  INTEGER KeyBuffer ! Key codes will be stored in this variable.
110  Keys$="ABC" !       Define keys that will initiate branches.
120  !
130  ON KYBD KeyBuffer,Keys$ GOSUB KBDService
140  !
150 Spin:  GOTO Spin
160  !
170 KBDService: ! Service routine for ON KYBD
180    IF KeyBuffer=NUM("A") THEN DISP "Alpha"
190    IF KeyBuffer=NUM("B") THEN DISP "Bravo"
200    IF KeyBuffer=67 THEN DISP "Charlie"
210  RETURN
```

The INTEGER statement declares a variable named KeyBuffer, which will be used as a one-keystroke buffer; in other words, when a key is pressed, the numeric code that it generates will be stored in this variable.

The Keys$ string variable is used to define which keys will be enabled to initiate a program branch. As each key is pressed, the string specified by Keys$ is searched for the presence of the corresponding code. If a match is found, then the branch is initiated; if not, the keystroke is ignored.

The ON KYBD statement enables branching to the subroutine called KBDService; the branch will be initiated whenever any of the keys that generate a code specified in the Keys$ variable is pressed. For instance, running the program and pressing an uppercase A will initiate a branch to the service routine.

The service routine can then determine which key was pressed by accessing the integer variable named KeyBuffer. This service routine defines different actions for pressing each key; pressing A results in the program displaying "Alpha"; pressing B results in "Bravo"; pressing C results in "Charlie".

Note that the key buffer contains the **numeric** code for the key, not the alphanumeric character that the key produces on the screen. Note also that pressing a key which generates a lowercase letter does **not** initiate a branch to the service routine.

Since the keyboard buffer is only one character in length, the service routine can miss keystrokes if keys are not processed quickly. This is due to the fact that keycodes are placed in the buffer as each keypress is detected; if a keycode is already there, then it is overwritten.

In order to disable certain key(s) from initiating the branch, you can execute an `OFF KYBD` statement that specifies the key(s). For example, this statement would disable **only** uppercase C from initiating the previously defined branch.

```
OFF KYBD "C"
```

Here is an example of disabling **all** keyboard branching:

```
OFF KYBD
```

With `ON KYBD`, you can also trap keystrokes which would otherwise cause immediate action. For example, most keyboards have a $\boxed{\text{Back space}}$ key that you can press to move the cursor left one space and erase the character at that location. Since this type of key produces an "escape sequence" (a sequence of characters beginning with the ASCII control character "escape"), you can include the key's escape sequence in the key string[1]. Here is an example:

```
320  BackSpace$=CHR$(27)&"D"
330  ON KYBD KeyBuffer,BackSpace$ GOSUB KBDService
```

The program segment places the escape code that is produced by the $\boxed{\text{Backspace}}$ key (on the Integral computer) into the variable named $\boxed{\text{Backspace}}$, and then enables the $\boxed{\text{Backspace}}$ key to initiate a branch by executing an `ON KYBD` statement.

Reading Text from the Screen

Somewhere between the high-level `INPUT` or `LINPUT` and the low-level `ON KYBD` statements lies another method of accepting alphanumeric input. The **`AREAD`** statement reads text from the screen into a string variable. See the example of using this statement in the preceding section of this chapter called "Sending Messages to the Operator".

[1] A list of the keys and the code that each produces is provided in the *Implementation Specifics* appendix to the *HP-UX Technical BASIC Language Reference*.

Using the Clock and Timers 10

Introduction

HP-UX systems feature a real-time clock that maintains date and time of day. You can access this clock from the Technical BASIC system[1]. There are also timers that allow you to generate interrupts at specified intervals.

Chapter Contents

This chapter covers using the clock and timers. Here are the task and topics covered:

Task/Topics	Page
Reading the current date	10-2
Reading the current time of day	10-2
Converting between various time and date formats	10-3
Measuring time intervals	10-6
Enabling timers to interrupt normal program flow at specified intervals	10-10

1 On multi-user HP-UX systems, *only* the system administrator can set the time and date.

Using the Clock

This section discusses the Technical BASIC features available for reading the date and time of day, and for measuring time between events.

Reading the Date

The DATE$ string function returns the current date in the form: *yy/mm/dd* (year/month/day).

```
DATE$
84/10/17
```

The numeric function for obtaining the date is DATE. Executing this function returns a date in the form: *yyddd* (*yy* indicates the last two digits of the year; *ddd* indicates the day of the year, in the range 1 through 366).

```
DATE
 84291
```

where the date displayed is the 291st day of the year 1984.

Reading the Time of Day

The TIME$ string function returns the system clock reading in this 24-hour notation: *hh:mm:ss* (hours:minutes:seconds). Assuming your system clock has been properly set, the reading returned by the TIME$ function shows the time elapsed since midnight of the current day.

```
TIME$
 8:54:47
```

Another method for determining the time elapsed since midnight is to use the TIME numeric function. This function returns the total number of seconds elapsed since midnight. This example of invoking the TIME function returns the numeric equivalent of a time of day of "8:54:57".

```
TIME
 32087
```

The last value returned in a day's time is 86 399. When the counter reaches this value, it is reset to 0 and the date is incremented by one. Note that all of the functions mentioned in this section are programmable. The following short program is an example:

```
10  DISP "Today's date is: "; DATE$;" or";DATE
20  DISP "This program was run at: ";TIME$
30  DISP "Time of day (in seconds since midnight) is: ";TIME
40  END
```

Time and Date Format Conversions

Technical BASIC has additional time functions that perform the following notational conversions:

- Converting a specified number of seconds (since midnight) to an hours:minutes:seconds (*hh:mm:ss*) string format.
- Converting a string in the form *hh:mm:ss* to the equivalent number of seconds (since midnight).
- Converting a specified Julian day number to a month/day/year (*mm/dd/yyyy*) string format.
- Converting a string in the form month/day/year (*mm/dd/yyyy*) to the equivalent Julian day number.

Time: Numeric to String Conversions

`HMS$` is a function which converts a specified number of seconds (since midnight) to an equivalent string in the form *hh:mm:ss* (hours:minutes:seconds). An example is as follows:

```
HMS$(TIME)
```

Here is what it might return:

```
09:45:52
```

Here is another example:

```
DISP "Elapsed time = ";HMS$(Time2-Time1)
```

In this case, the time returned would not be in seconds since midnight; however it would be in a more usable form than just seconds:

```
10:10:00
```

Thus, the elapsed time is 10 hours, ten minutes, and no seconds.

Time: String to Numeric Conversions

The HMS function does the opposite of HMS$. This function converts a string in the form *hh:mm:ss* (hours:minutes:seconds) to the integer equivalent in seconds. Here is an example:

```
100  ! Calculate time differential
110  ! and display in "hh:mm:ss" format.
120  !
130  DISP "Time between  6:08:29 p.m."
140  DISP "         and 10:14:32 a.m."
150  !
160  ! Now calculate difference (in minutes).
170  Diff=HMS("12:00:00")+HMS("06:08:29")-HMS("10:14:32")
180  ! Then re-format for human consumption.
190  Diff$=HMS$(Diff)
200  !
210  DISP "          is ";Diff$
220  !
230  END
```

Here are the program's results.

```
Time between  6:08:29 p.m.
         and 10:14:32 a.m.
          is  7:53:57
```

The HMS function can be also be executed from the keyboard:

```
HMS("13:30:15")
```

which returns on the display:

```
48615
```

Date: Numeric to String Conversions

The MDY$ string function converts a Julian Day number[1] to an equivalent string expression in the form: *mm/dd/yyyy* (month/day/year). The range of Julian Day numbers that you can pass to this function is from 2299161 through 3199160; these limits correspond to October 15, 1582[2] and November 25, 4046, respectively.

Here is an example of how you can use MDY$:

```
MDY$(2446000)
```

The function returns the Julian Day number in a more understandable format.

```
10/26/1984
```

[1] The Julian Day number is an astronomical convention representing the number of days since January 1, 4713 B.C.
[2] The beginning date of the modern Gregorian calendar.

Date: String to Numeric Conversions

The MDY numeric function does the opposite of the MDY$ function. When it is given a string in the form *mm/dd/yyyy*, it returns the equivalent Julian Day number. Note that the string must lie between the dates 10/15/1582 and 11/25/4046, and consist of exactly 10 characters (including the two slashes). Here is an example of using the function from the keyboard:

```
MDY("11/25/4046")
```

returns the following on the display:

```
3199160
```

Here is a more current example:

```
MDY("10/17/1984")
```

returns the following on the display:

```
2445991
```

Timing the Interval Between Events

Measuring the time between two events is quite simple.

```
100  Tinit=TIME !      Initial time.
110  !
120  FOR J=1 TO 5555
130  !
140  NEXT J
150  !
160  Tfinal=TIME !     Final time.
170  !
180  DISP "Elapsed time =";Tfinal-Tinit;"seconds."
190  !
200  END
```

Here are typical results of running the program:

```
Elapsed time = 15 seconds.
```

Note that the program does not keep track of changes in the day. Thus, if you are timing events that will occur near midnight, you may get a negative time interval. You may want to add code that keeps track of days also. For example, you could multiply the difference in days by the number of seconds in a day (86 400), and add this figure to the differential.

Using the Timers

This section covers the following timer operations:

- Timer branches.
- Measuring time elapsed since a timer was set to interrupt.

Timer Interrupts

The subject of event-initiated branching was discussed near the end of the "Program Structure and Flow" chapter. If you are not familiar with the concept, you may want to read that section before reading this section.

Here are the statements that control timer-initiated branches:

- `ON TIMER#` sets a specified timer to zero and immediately activates it. The end-of-line branch is initiated when the specified time interval has elapsed. Three timers are available for this purpose; they are numbered 1, 2, and 3.
- `OFF TIMER#` disables branching for the specified timer.

You can use these timers to generate these types of interrupts:

- Cyclic interrupts
- Delay interrupts
- Time-of-day interrupts

Cycle and Delay Interrupts

The `ON TIMER#` statement enables a branch to be taken as soon as the specified number of *milliseconds* have elapsed. For instance, the following statement enables a `GOSUB` branch to the subroutine called Cycle2 to occur two seconds from the time that the statement is executed:

```
ON TIMER# 1, 2000 GOSUB Cycle2
```

`ON TIMER#` remains in effect, re-initiating a branch every two seconds until an `OFF TIMER#` statement is executed (for timer number 1). Thus, the `ON TIMER#` statement creates a cyclic interrupt.

To produce a one-time timer interrupt (i.e., a delay interrupt), you will need to execute an `OFF TIMER#` statement in the timer service routine.

This example shows both usages of timers. It displays the time (hours:minutes:seconds) for a period of two seconds and then prints five random numbers. It repeats this process until eight seconds have elapsed, at which time the program is ended. The `ON TIMER# 1` is a cyclic interrupt, while the `ON TIMER# 2` statement, along with its `OFF TIMER# 2` counterpart, act as a one-time delay interrupt.

```
100  ON TIMER# 1,2000 GOSUB TwoSecCycle
110  ON TIMER# 2,8000 GOTO EightSecDelay
120  !
130  CLEAR !  Clear screen.
140  !
150  DummyLoop: ALPHA 5,1 @ DISP TIME$
160             GOTO DummyLoop
170             !
180  TwoSecCycle: ALPHA 8,1
190               FOR Number=1 TO 5
200                 DISP RND ! Random number.
210               NEXT Number
220               !
230  RETURN
240  !
250  EightSecDelay: OFF TIMER# 2
260                 ALPHA 15,1
270                 DISP "Finished"
280                 !
290  END
```

Here is typical output from the program.

```
12:20:25


 .744804223761712
 .0289620654927213
 .559984130375072
 .311563463240455
 .114398065498712


Finished
```

Simulated Time-of-Day Interrupts

The ON TIMER# statement allows you to define and enable a branch to be taken when the timer reaches a specified count. You can simulate time-of-day interrupts by using this procedure:

1. Determine the current time of day.
2. Determine the desired time of day at which the interrupt will occur.
3. Calculate the number of seconds between the two.
4. Set a timer interrupt for that number of seconds (from the present time).

Typically, the ON TIMER# statement is used to cause a branch at a specified time. This statement can be use as an interval timer in a program, by storing in a program variable the value of the system clock when the program is started (using the function called TIME) and subtracting this value from a specified final time. The following example uses the interval timer as an alarm to remind you to go to lunch.

```
100  DISP "This is the present time of day: ";TIME$
110  DISP
120  DISP "Specify alarm time using this format: 'hh:mm:ss'"
130  LINPUT Tfinal$ @ DISP "Thank you."
140  !
150  ! Determine number of seconds since midnight.
160  Tfinal=HMS(Tfinal$)
170  ! Set timer to interrupt in Tfinal-TIME (seconds).
180  ON TIMER# 1,(Tfinal-TIME)*1000 GOSUB Alarm
190  !
200  Spin: GOTO Spin !  Twiddle thumbs.
210  !
220  Alarm:
230      BEEP
240      CLEAR
250      DISP @ DISP @ DISP "Time for lunch!" @ DISP
260      OFF TIMER# 1
270      RETURN
280      !
290  END
```

Here are is the screen that the program produces:

```
This is the present time of day:  9:38:54

Specify alarm time using this format:  'hh:mm:ss'
? 11:45:00
Thank you.
```

The first line of output gives the current time of day. The second line asks you to set the alarm for a time of your choosing using the 'hh:mm:ss' format. The string you enter (11:45:00 above) is converted by the numeric function `HMS` into seconds since midnight and assigned to the variable Tfinal. Next, timer number 1 is set to interrupt; the time interval is calculated as the difference between "Tfinal" and the current `TIME` (it is multiplied by 1000 to convert the result to milliseconds, which is how the `ON TIMER#` statement interprets the interval parameter).

When the specified time interval has elapsed, the timer interrupt service routine displays the "lunch alarm" message.

```
Time for lunch!
```

Timer Functions

The `READTIM` numeric function returns the number of seconds currently registered on the specified system timer.

- For timer numbers 1, 2, or 3, this is the number of seconds (**not** milliseconds) since the timer was set in the program, or since it last initiated a branch.
- For timer 0, it is the number of seconds elapsed since the system clock was last set, either by the system administrator or by power on.
- If the timer is not currently being used, then `READTIM` returns 0.
- After an `OFF TIMER#` statement, `READTIM` returns the reading of the timer at the point it was disabled.

The following program makes use of the `READTIM` function. It programmatically defines a function key to call a routine which displays the number of seconds elapsed since timer 1 was set. After ten minutes have elapsed, it displays the message:

```
Ten minutes have elapsed.
```

and then issues a beep.

```
100 ON TIMER# 1,10*60*1000 GOSUB TenMin ! Interrupt after 10 min.
110 ON KEY# 1,"Seconds" GOSUB Elapsed !   Show elapsed time.
120 !
130 Spin: GOTO Spin !                     Idle loop.
140 !
150 STOP
160 !
170 TenMin: DISP "Ten minutes have elapsed."
180         BEEP
190   RETURN
200   !
210 Elapsed: DISP READTIM(1);"seconds since timer 1 set."
220   RETURN
```

Pressing $\boxed{\text{f1}}$[1] directs the program to display the number of seconds since TIMER# 1 was set. Here are typical results that the Elapsed subroutine displays:

```
3 seconds since timer 1 set.
4 seconds since timer 1 set.
9 seconds since timer 1 set.
```

Timers and Subprograms

It is possible for a context (program or subprogram) to enable a timer interrupt and then call one or more subprograms before the timer interrupt occurs. As long as the context is *not* executing a subprogram *when* the timer is expected to interrupt, the interrupt will initiate its branch at the correct time. However, if the subprogram *is* being executed when the timer would have otherwise initiated its branch, then the branch to the service routine is not executed until *after* control returns to the context that defined the timer interrupt.

Timer Interrupts While Not Executing a Subprogram

The following program is an example of the situation in which the subprogram is finished before the timer interrupts. (The situation of the subprogram being executed when the calling context's timer interrupt would have occurred is covered in the next section).

```
100 ON TIMER# 1,10000 GOSUB TenSecs !  10-second cycles.
110 !
120 Tinit=TIME !  Store initial time.
130 !
140 FOR I=1 TO 3 !  Wait 3 seconds.
150   WAIT 1000 @ DISP TIME-Tinit;"seconds."
160 NEXT I
170 !
180 CALL "SUBTimer1" (Tinit)
190 !
200 FOR I=1 TO 3 !  Wait 3 more seconds (to allow interrupt).
210   WAIT 1000 @ DISP TIME-Tinit;"seconds."
220 NEXT I
230 !
240 END
250 !
260 TenSecs: DISP
270          DISP "At branch to 'TenSecs', READTIM(1)=";READTIM(1)
280          DISP
290   RETURN
```

1 On some consoles, this key is labeled $\boxed{\text{k0}}$. Refer to the *Getting Started Guide* for your particular Technical BASIC system for a description of ON KEY# parameters and softkey labels.

Here is the subprogram.

```
100  SUB "SUBTimer1" (Tinit)
110  DISP
120  DISP "Entering SUBTimer1."
130  DISP
140  FOR I=1 TO 5
150    WAIT 1000
160    DISP TIME-Tinit;"seconds."
170  NEXT I
180  DISP
190  DISP "Exiting SUBTimer1."
200  DISP
210 SUBEND
```

Here are the results of running this program.

```
  1 seconds.
  2 seconds.
  3 seconds.

Entering SUBTimer1.

  4 seconds.
  5 seconds.
  6 seconds.
  7 seconds.
  8 seconds.

Exiting SUBTimer1.

  9 seconds.
  10 seconds.

At branch to 'TenSecs', READTIM(1)= 0

  11 seconds.
```

The main program starts out by setting timer number 1 to interrupt in ten seconds. The elapsed time is then read every second and displayed until branching to the subprogram.

The subprogram displays a message telling you that it has been given control. It also displays elapsed times every second (for 5 seconds). After five seconds have elapsed, control is returned back to the calling program.

When timer 1 has counted to 10 seconds, the branch to TenSecs is initiated.

The **main point** of this example is that the main program's timer interrupt occurs at the expected time, because the subprogram is not being executed **when** the timer interrupts.

The program also shows that the timer is reset to 0 (as determined by the READTIM function); however, it does not show that the timer is cyclic and is automatically re-enabled and begins counting again. In this case, the program ended before a second interrupt occurred.

Timer Interrupts while Executing Subprograms

The following program and subprogram show an example in which the subprogram is being executed when the calling context's timer interrupts. Note that they are slightly modified versions of the preceding program and subprogram.

```
100 ON TIMER# 1,10000 GOSUB TenSecs !  10-second cycles.
110 !
120 Tinit=TIME !  Store initial time.
130 !
140 FOR I=1 TO 3 !  Wait 3 seconds.
150   WAIT 1000 @ DISP TIME-Tinit;"seconds."
160 NEXT I
170 !
180 CALL "SUBTimer2" (Tinit)
190 !
200 FOR I=1 TO 11 !  Wait 11 more seconds (to allow interrupt).
210   WAIT 1000 @ DISP TIME-Tinit;"seconds."
220 NEXT I
230 !
240 END
250 !
260 TenSecs: DISP
270          DISP "At branch to 'TenSecs', READTIM(1)=";READTIM(1)
280          DISP
290   RETURN
```

Here is the subprogram.

```
100  SUB "SUBTimer2" (Tinit)
110  DISP
120  DISP "Entering SUBTimer2."
130  DISP
140  FOR I=1 TO 10
150    WAIT 1000
160    DISP TIME-Tinit;"seconds."
170  NEXT I
180  DISP
190  DISP "Exiting SUBTimer2."
200  DISP
210 SUBEND
```

Here are the results of running this program.

```
    1 seconds.
    2 seconds.
    3 seconds.

   Entering SUBTimer2.

    4 seconds.
    5 seconds.
    6 seconds.
    7 seconds.
    8 seconds.
    9 seconds.
    10 seconds.
    11 seconds.
    12 seconds.
    13 seconds.

   Exiting SUBTimer2.


   At branch to 'TenSecs', READTIM(1)= 3

    14 seconds.
    15 seconds.
    16 seconds.
    17 seconds.
    18 seconds.
    19 seconds.
    20 seconds.

   At branch to 'TenSecs', READTIM(1)= 0

    21 seconds.
    22 seconds.
    23 seconds.
    24 seconds.
```

The main program starts out by setting timer number 1 to interrupt in ten seconds. The elapsed time is then read every second and displayed until branching to the subprogram.

The subprogram displays a message telling you that it has been given control. It also displays elapsed times every second (for 10 seconds this time). After ten seconds have elapsed, control is returned back to the calling program.

The timer in the main program would have initiated its branch, but could not because the subprogram was being executed. This result is shown by the value 3 being returned by the `READTIM` function. In the calling context (here the program), timer 1 did count to 10 seconds, but it could not initiate the branch to TenSecs because it was not in the current context (the subprogram).

The **main point** of this example is that the main program's timer interrupt is **delayed**, because the subprogram does not return control to the calling context (main program) until **after** the timer interrupt should have occurred. However, the branch is initiated as soon as control returns to the context in which it is enabled.

The program executes 10 additional 1-second waits, in order to demonstrate that the timer will indeed initiate subsequent branches as expected.

Data Storage and Retrieval 11

Introduction

This chapter describes some useful techniques for storing and retrieving data. The methods fall into these categories:

- Storing data with programs (using `DATA` and `READ` statements)
- Storing data in BASIC/DATA files (using `ASSIGN#`, `PRINT#`, and `READ#`)
- Storing data in text/data files (using `ASSIGN`, `OUTPUT`, and `ENTER`)

To store and retrieve data that is part of the BASIC program, use `DATA` statement(s) to specify data that is to be stored in the memory area used by BASIC programs; thus, the data is always kept in the same file as the program. 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 BASIC/DATA files are more appropriate. These files provide means of storing data on mass storage devices. The BASIC/DATA files available with Technical BASIC are described in this chapter. A number of different techniques for accessing data in these files are described in detail.

Files of type text/data are used as the interchange method for sharing data between Technical BASIC and the HP-UX system.

Chapter Contents

This chapter discusses these topics:

Tasks/Topics	Page
Storing data in programs	11-3
Using data files	11-9
Brief mass storage tutorial	11-9
Introduction to BASIC/DATA file access techniques	11-11
A closer look at file access	11-14
A closer look at serial access	11-20
Random file access	11-24
Determining data types	11-28
Trapping EOF and EOR conditions	11-29
Using text/data files	11-31

Storing Data in Programs

This section describes a number of ways to store values in memory. In general, these techniques involve using program variables to store data. The data are kept with the program when it is stored on a mass storage device (with STORE and SAVE). These techniques allow extremely fast access of the data. They provide good use of the computer's memory for storing relatively small amounts of data.

Storing Data in Variables

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 was used in the first example program in the "Program Development" chapter. It was a convenient way to store data without knowing anything about data files.

```
     .
     .
     .
110  OPTION BASE 1
120  DIM IncomeName$(2)
130  REAL TargetIncome(2)
140  !
150  ! Assign values to variables.
160  LET IncomeName$(1)="Payroll"
170  LET IncomeName$(2)="Investments"
180  LET TargetIncome(1)=1680.00
190  LET TargetIncome(2)=345.67
     .
     .
     .
```

This technique works well when there are only a relatively few "constants" 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 one 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   DISP "Please enter your ID, and press Return."
110   INPUT ID
  .
  .
210   LINPUT "Enter the value of X",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, limited only by computer memory (or disc space when the program is stored in a file).

Storing Data

When you `RUN` a program, the system concatenates all `DATA` statements in a given 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 Payroll,Investments
       .
       .
200 DATA 1680.56,345.67
       .
       .
300 DATA Mortgage
```

Payroll	Investments	1680.56	345.67	Mortgage

Figure 11-1. Data Stream

As you can see from the example above, a data stream can contain both numeric and string data items.

Each data item must be separated by a comma; string items can optionally be enclosed in quotes. Strings that contain a comma or exclamation mark must be enclosed in quotes. In addition, you must use the following notation for every quote you want in the string. For example, to enter the strings UNQUOTED, UNQUOTED, and "QUOTED" into a data stream, use this DATA statement:

```
100 DATA UNQUOTED,"UNQUOTED","~"QUOTED~""
```

The tilde characters indicate that the quote mark that follows it is to be part of the data read into a string variable.

Retrieving Data

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. Here is an example:

```
100 DATA Payroll,Investments
        .
        .
200 DATA 1680.56,345.67
        .
        .
100 READ Income1Name$,IncomeName2$,TargetIncome1
```

This **READ** statement would read three data items from the data stream into the three variables. Note that the first and second variables are string and the third is a numeric. This corresponds to the order and type of data items in the data stream.

Numeric data items can be **READ** into either numeric or string variables, with the following restrictions:

- If the numeric data item is of a different specific numeric type than the numeric variable, then the item is automatically converted. For instance, **REAL**s are converted to **INTEGER**s, and **INTEGER**s to **REAL**s. However, if the value is out of range for that numeric data type, then an error is reported.
- If the string variable has not been dimensioned to a size large enough to hold the entire data item, then error 56 is reported.

The Data Pointer

The system keeps track of which data item to READ next by using a 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 initially points 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. When a subprogram is called by a main program (or another subprogram), 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 assigned a value. 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 1984,"The date is "
40  READ Str$
50  Print Str$;Month$;Day;Year$
60  END

The date is November 26 1984
```

Storage and Retrieval of Arrays

In addition to using READ to assign values to string and numeric variables, you can also READ data into arrays. The system will match data items with variables one at a time until it has filled a row. The next data item then becomes the first element in the next row. You must have enough data items to fill the array or you will get an error. In the example below, we show 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  MAT READ Example
50  MAT PRINT USING "3(X,K),/";Example
60  END
RUN

 1 2 3

 4 5 6

 7 8 9
```

The data pointer is left at item 10; thus, items 10 and 11 are saved for the next READ statement.

Moving the Data Pointer

In some programs, you will want to assign the same data items to different variables. To do this, you have to move the data pointer so that it is pointing at the desired data item. You can accomplish this with the `RESTORE` statement. If you don't specify a line number or label, `RESTORE` returns the data pointer to the first data item of the first data stream in that context. If you do include a line identifier in the `RESTORE` statement, the data pointer is moved to the first data item in the `DATA` statement at the identified line. The example below illustrates how to use the `RESTORE` statement.

```
100   DIM Array1(2)      ! 3-element array (OPTION BASE 0).
110   DIM Array2(4)      ! 5-element array (OPTION BASE 0).
120   DATA 1,2,3,4
130   DATA 5,6,7
140   READ A,B,C         ! Reads first 3 items in stream.
150   MAT READ Array2    ! Reads next 5 items in stream.
160   DATA 8,9
170                      !
180   RESTORE            ! Re-positions pointer to 1st item (line 120).
190   MAT READ Array1    ! Reads first 3 items in stream.
200   RESTORE 160        ! Moves data pointer to item "8".
210   READ D             ! Reads "8".
220                      !
230   PRINT "Array1 contains:"
240   MAT PRINT Array1
250   PRINT "Array2 contains:"
260   MAT PRINT Array2
270   PRINT "A,B,C,D equal: ";A;B;C;D
280   END
```

Here are the results of running the program.

```
Array1 contains:
  1
  2
  3
Array2 contains:
  4
  5
  6
  7
  8
A,B,C,D equal:  1  2  3  8
```

Using BASIC/DATA Files

This section of this chapter describes the another general class of data storage and retrieval methods — that of using mass storage BASIC/DATA[1] files. This material is broken up into several parts.

- A look at mass storage, directories, and BASIC/DATA files
- Introduction to accessing BASIC/DATA files
- A closer look at using files
- Determining data types
- Trapping EOF and EOR conditions

Brief Mass Storage Tutorial

This section briefly discusses these topics:

- Mass storage in general
- Directories
- BASIC/DATA files

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. However, most mass storage devices are capable of storing on the order of hundreds of thousands to several million data 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 known as *files*; a file is merely a collection of data items which are accessed through one name. Each file may contain one or more *logical records*; each logical record in a file is much like a subset of the file in that it can also contain several data items.
- Files are organized by directories. A directory is an index of files; in any directory, there is an entry for every file within that directory.

1 The subsequent section called "Using text/data Files" discusses techniques for accessing files of type **text/data**, which is the file type that both Technical BASIC and HP-UX can use (such as for data interchange).

When a data file is initially created, it contains nothing. However, you can fill it with any data that you want, which gives the file the general structure shown below.

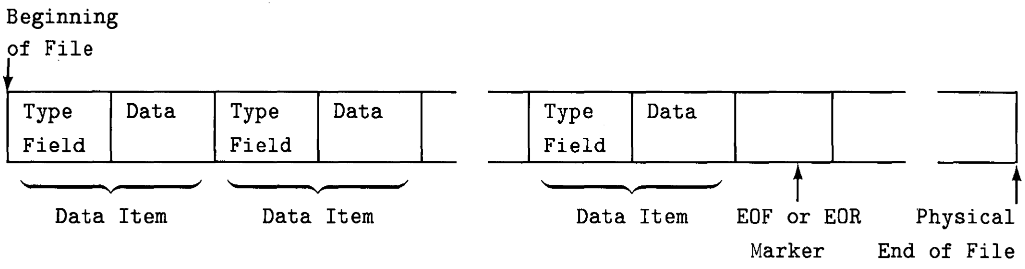


Figure 11-2. General Structure

The data items are stored using either ASCII characters (for string items) or an internal representation (for numeric items). The type fields indicate whether the item is a string or a numeric item. Subsequent sections provide further details of just what the file contains and how to write to and read from them.

The `CAT` statement shows some of the information that is stored in a directory. Executing `CAT` with no directory path tells the system to get a catalog of the *current working directory*[1].

```
CAT
```

Specifying a directory path with the file name gives a listing of the files in that directory.

```
CAT "/users/mark/BASICFILE"
```

[1] If you don't know the meaning of the term "current working directory," then refer to the discussion of the HP-UX file system in your HP-UX system's documentation.

Introduction to File Access Techniques

This section presents BASIC programming techniques useful for accessing BASIC/DATA files. The first section gives a brief introduction to the steps you might take to store data in a file. Subsequent sections describe further details of these steps.

Methods of Accessing Data Files

There are two methods of accessing BASIC/DATA files:

- **Serial access:** writing to or reading from the file in sequential order — one item at a time, from the beginning.
- **Random access:** writing to (or reading from) the file, starting at the beginning of any *logical record* within the file. Within any logical record, however, access is strictly serial.

Technical BASIC allows you to use both types of access methods on one file, with only a few restrictions. Each access method has uses in certain applications.

Example of Writing Serially to a File

Storing data in files requires a few simple steps. The following program segment shows a simple example of placing several items in a data file.

```
100  ! Allocate memory for variables.
110  OPTION BASE 1
120  DIM IncomeName$(2)
130  REAL TargetIncome(2)
140  !
150  ! Assign variables.
160  IncomeName$(1)="Payroll"
170  IncomeName$(2)="Investments"
180  !
190  TargetIncome(1)=1680.56
200  TargetIncome(2)=345.67
210  !
220  ! Create a data file.
230  CREATE "Oct84Income",1 !      Size = 1 logical record.
240  ASSIGN# 1 TO "Oct84Income" !  Assign a buffer to it.
250  !
260  PRINT# 1 ; IncomeName$(1),TargetIncome(1)
270  PRINT# 1 ; IncomeName$(2),TargetIncome(2)
280  !
290  ASSIGN# 1 TO "*"  !           Close file (release buffer).
300  !
310  END
```

In order to store data in a file, a data file must be created (or already exist) on the mass storage media to be used. In this case, line 230 creates a BASIC/DATA file for storage. The file is created with 1 *logical* record, which has a default size of 256 bytes. This is a large enough file to store the data in this example. (File size, logical records, and record size are discussed in the subsequent section called "A Closer Look at File Access".)

The file is created in the "current working directory." If the file is to be created in another directory, then the appropriate directory path must be prefixed to the file name. This example creates a file in another directory:

```
CREATE "/users/mark/NovIncomes",4
```

See the *Getting Started* manual for your particular Technical BASIC system for specific information about directories on your system.

Then, in order to store data in (or retrieve data from) the file, you must assign a *buffer number* to the file. Line 240 shows an example of assigning a buffer number to the file (also called "opening the file"). The `PRINT#` statements on lines 260 and 270 send the previously defined data items being sent to the file.

The file is closed after all data have been sent to the file. (In this case, the close operation is not necessary, because all files are automatically closed by the system by the `END` statement.)

NOTE

This is a closed data file and cannot be retrieved/read with `GET` or `LOAD`. To retrieve/read the data use the program shown next.

Here is a conceptual diagram of the file's contents after the program has finished execution.

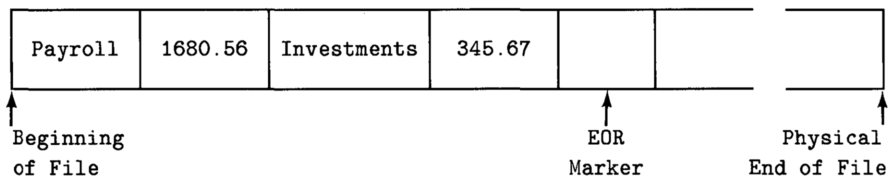


Figure 11-3. File's Contents

Although they are not shown in the drawing, the system automatically adds the type fields. You can use the TYP function to read it from BASIC and thus determine the item's type. The subsequent section called "Determining Data Types" gives further details.

The end-of-record (EOR) marker is always placed after the last item written into a file. It is used instead of an end-of-file (EOF) marker, because Technical BASIC allows both random and serial access of the same file. Note, however, that the file is initially filled with EOF markers (when the disc is initialized). Subsequent sections explain EOR and EOF markers in greater depth.

Example of Serially Reading from a File

Here is a simple program that reads the data stored in the file created and written in the preceding example.

```
100  ! Allocate memory for variables.
110  OPTION BASE 1  !  Implicit lower subscript bound.
120  DIM IncNam$(2)
130  REAL TgtInc(2)
140  !
150  ASSIGN# 1 TO "Oct84Income" !  Assign buffer.
160  !
170  READ# 1;IncNam$(1),TgtInc(1) ! Read 2 items from file.
180  READ# 1;IncNam$(2),TgtInc(2) ! Read 2 more items.
190  !
200  DISP
210  DISP "  Category              Target"
220  DISP " ----------            --------"
230  DISP IncNam$(1),TgtInc(1)
240  DISP IncNam$(2),TgtInc(2)
250  !
260  ASSIGN# 1 TO "*"
270  !
280  END
```

As in the preceding example, you must assign a file number to the data file before you can access it. Line 150 makes this assignment.

The subsequent READ# statements (lines 170 and 180) read the data into program variables. The general suggestion is to **"read it like you wrote it"**; in other words, match the order and type of each item in the file to the variable into which the item will be read. For instance, if you wrote a SHORT variable, a REAL numeric expression, and a string of length 36 characters into the file, then you should read these items using a SHORT variable, a REAL variable, and a string variable with a length of (at least) 36 characters.

NOTE

The variable used to read each data item need only be of the same *general* data type as the data item (i.e., numeric or string). It is not strictly required to be of the same *specific* type (i.e., INTEGER, SHORT, or REAL for numeric items; or identical length for string items). However, if the specific types of variables and items are not matched, it becomes possible to generate range errors; for instance, a value of 1E+200 is out of range for INTEGER and SHORT variables, and a string of 200 characters is out of range for a string variable with maximum length of 18 characters.

As you can see, these are very simple examples. However, they show the general steps you must take to serially access files.

A Closer Look at BASIC/DATA File Access

The preceding section showed simple examples of writing data in a file and then reading it back. This section describes what is happening "behind the scenes." It will help you to better understand how to create and use BASIC/DATA files.

File and Record Size Calculations

In the preceding example of serially writing a file , the following program line created a file that was used to store the program's data:

```
230 CREATE "Oct84Income",1 !   Size = 1 logical record.
```

The size of the file was specified to be one logical record with (default) record size of 256 bytes. The example stated that this file size was sufficient to store the data. This section will help you verify that statement and show you how to calculate the size of file required to store any data.

The following chart describes the amount of space necessary to store numeric and string data items.

Table 11-1 Data Storage Requirements

Data Type	Storage Requirements (Per Data Item)
Simple REAL numbers	1 byte for type field, + 8 bytes for number.
Simple SHORT numbers	1 byte for type field, + 8 bytes for number.
Simple INTEGERs	1 byte for type field, + 4 bytes for number.
Simple string	3 bytes for type field, + 1 byte per character, + 3 bytes each time the string crosses a logical record boundary.
REAL array (each element)	1 byte for type field, + 8 bytes for number.
SHORT array (each element[1])	1 byte for type field, + 4 bytes for number.
INTEGERarray (each element)	1 byte for type field, + 4 bytes for number.
String array	Each array element requires the same amount of space required for simple string items (shown above)

Using the data from the preceding serial access example, here are the calculations for the size of file required to store the data:

Table 11-2 Storage Calculations

Item	Type of data	Bytes required
IncomeName$	"Payroll"	7+3 = 10
	"Investments"	11+3 = 14
TargetIncome	Two REAL numbers	2*(1+8) = 18
		Total = 42

[1] If an entire **SHORT** array is written into the file as an array variable, then each element requires 5 bytes; however, if **SHORT** array elements are written individually, then each element requires 9 bytes.

The file size could be 1 logical record (with default size of 256 bytes). You would not need to partition it into smaller logical records, since the data items in the file are only accessed serially.

Note that the size of the file actually created will always be an integral multiple of 1 024 bytes. This effect is due to the fact that the HP-UX file system can only address portions of the disc as small as a 1 024-byte block. Blocks and records are discussed next.

Records and Blocks

A *record* or *block* is the smallest unit of mass storage space that is *independently addressable*. There are three types:

- **Logical records** are the smallest unit of mass storage that can be addressed *by a BASIC program*. You can specify the size of logical records in a file when you execute a `CREATE` statement. If no logical record length is specified, a length of 256 bytes is assumed.
- **Blocks** are the smallest unit of mass storage that can be handled *by the HP-UX file system*. HP-UX file system blocks are always 1 024 bytes in length.
- **Physical records** are the smallest unit of storage that can be addressed *by a mass storage device*. With most HP disc drives, physical records are 256, 512, or 1 024 bytes in length.

Logical records make it possible to partition a file into several smaller units, each of which the BASIC system can address independently. In fact, each logical record is similar to a file in the respect that it is independently addressable. Within any file, all logical records are the same length; however, each file may have a different logical record length.

Blocks are mentioned only so that you will understand why a file with length of 1 024 bytes will be created if you try to create a file with a length of 256 bytes.

Physical records are only mentioned to avoid confusion with logical records and blocks, should you happen to see that term in your disc manual.

When you create a data file, you specify these parameters: file name, number of logical records, and logical record length (optional). The following drawing shows the file that is created by this statement:

```
CREATE "File_xyz",1,300
```

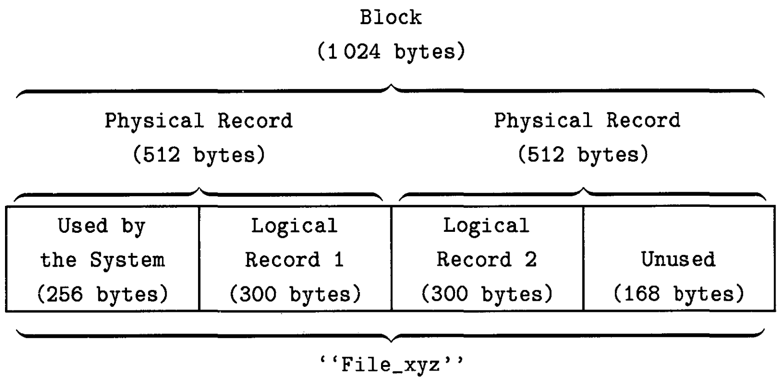


Figure 11-4. Data File Parameters

The example shows several important points about files.

- The file takes up 1 024 bytes of storage, since a file always contains an integral number of blocks. (Similarly, files always begin on a physical record boundary, and thus always contain an integral number of physical records.)
- The Technical BASIC system always uses the first 256 bytes of a BASIC/DATA file for keeping information such as logical record size, number of records, etc.
- After allocating the first 256 bytes for overhead, the system allocates logical records. The first logical record begins at the byte following the last byte of system overhead, and the second record begins on the byte just following the last byte of the first logical record.

  The example also shows that the system will allocate more logical records than specified, if there is room in the file. In this case, there was enough room for one more logical record. As another example, if you create a file with 1 logical record of length 256 bytes, then the file will actually contain 3 records; the system allocates two additional records, rather than leaving the last 512 bytes unusable.
- There are 168 bytes of unusable space at the end of this example file (1 024−256−2∗300), because the next file begins on the next block boundary (which also aligns with the physical record boundary).

The File Pointer

The system uses a file pointer to locate and access the data items in a file. The file pointer points to the place where data will be:

- written with the next `PRINT#` statement, or
- read with the next `READ#` statement.

The file pointer is updated automatically by the system whenever the file is accessed. More information about the file pointer will be given in subsequent examples.

File Buffers

When a buffer number is assigned to a file, such as in the following statement:

```
ASSIGN# 2 TO "Oct84Income"
```

the BASIC system sets up a file buffer through which it communicates with the mass storage device. This buffer is a small portion of your BASIC memory area, usually a few hundred bytes in length.

Here is a pictorial representation of the communication path.

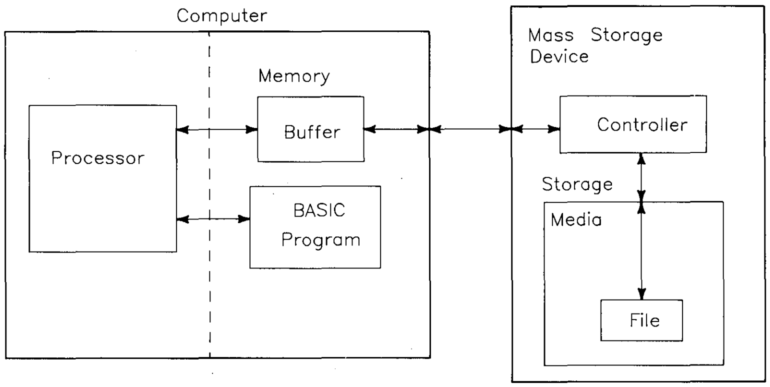


Figure 11-5. File Buffer

The purpose of the buffer is to decrease access time for information and reduce the wear on the physical mass storage devices.

Here is an example of how a buffer works. Assume the following conditions: you have created a file with logical records of 9 bytes each, and you want to access 20 of these records in a short program segment.

Without buffering, the BASIC system would have to make 20 different accesses of a mass storage device to obtain the information. And each time an item is requested from the mass storage device, the BASIC system would get a whole block (1 024 bytes) of information, since that is the smallest unit of data that the HP-UX file system can address. Considering the possibility that all of these items might all be located in the same 1 024-byte block, the system would, in this case, be getting about 100 times the information it needs in each of 20 separate mass storage accesses.

With buffering, the BASIC system loads a physical record of information from the mass storage device, and then extracts from that record the information it needs. In our example, if all 20 logical records are in the same mass storage block, the computer only has to make 1 mass storage access; it then can extract each logical record *from the buffer*. Overall, in this particular example, the amount of disc access has been reduced by a factor of 20, and the information flow has been reduced by a factor of about 2000 (=100*20).

This example is *not* necessarily representative of how much mass storage wear and access time can be saved by buffering, but it does make the point that buffering is generally a good technique to use.

File buffers are *automatically* sent to the mass storage device (while writing) at the following times:

- Whenever the buffer gets full, or when data items in another block are accessed.
- When the file is closed (or when the file number is re-assigned).
- When the program is halted (i.e., when `PAUSE`, `STOP`, or `END` is executed).
- When program execution is interrupted (by an event that is set up to cause an "event-initiated branch", as described in the "Program Structure and Flow" chapter).
- When a `PRINT#` statement is executed *from the keyboard*.

A Closer Look at Serial Access

Serial access is used when a quantity of data is to be stored sequentially in a file and then read back in the same (sequential) order. With this type of access, the file itself is the smallest addressable unit of storage. This is true even if the file being accessed consists of more than one logical record, because the data items are stored and retrieved without regard to logical record divisions (during serial access).

Serial Write Operations

When a file is opened, the file pointer is placed at the beginning of the file.

```
ASSIGN# 1 TO "BudgetData"
```

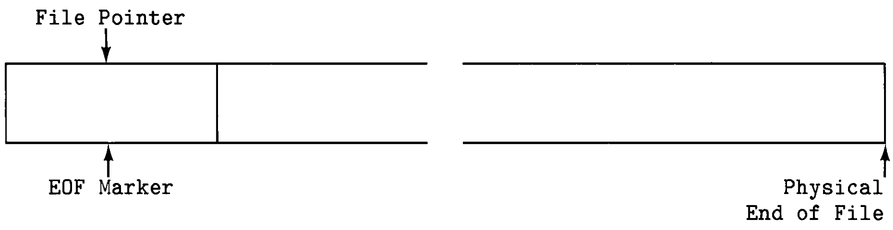


Figure 11-6. File Initially

The drawing shows that the file initially contains an end-of-file (EOF) marker at the beginning of the file. Actually, the file is entirely full with EOF markers at the point the file is created.

When a `PRINT#` statement writes data into the file (through the buffer assigned to the file), the data items are sent one at a time, from left to right in the list, starting at the location indicated by the file pointer. As each item in the data list is stored, the pointer is updated to point to the next available location. When all items in the list have been recorded, the file pointer points at a location just past the end of the recorded data. An end-of-*record*[1] (EOR) marker indicates the position of the last recorded data item.

[1] An EOR marker is used instead of an EOF marker, because you can randomly and serially access a file.

The location of the file pointer is the point at which a subsequent PRINT# statement will begin.

```
PRINT# 1;IncomeName$(1),TargetIncome(1)
```

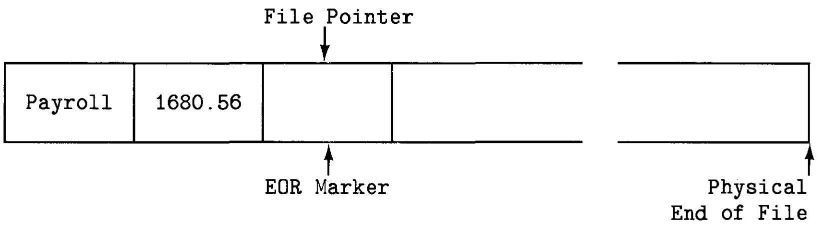


Figure 11-7. File Pointer Location 1

Execution of a subsequent PRINT# statement to the same buffer records the items in the corresponding data list beginning at the current file pointer. The system overwrites the existing EOR marker, writes the items (and corresponding type fields), and then writes another EOR marker at the end of this newly recorded data.

```
PRINT# 1;IncomeName$(2),TargetIncome(2)
```

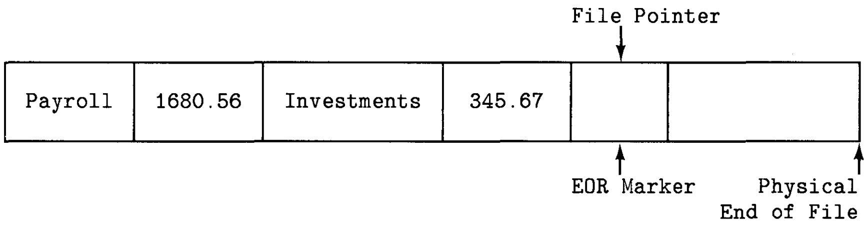


Figure 11-8. File Pointer Location 2

Earlier in the chapter, it was stated that serial writing essentially ignores logical record boundaries. Here is what actually happens when a serial PRINT# statement crosses a logical record boundary.

```
PRINT# 1;12.05,"String data"
```

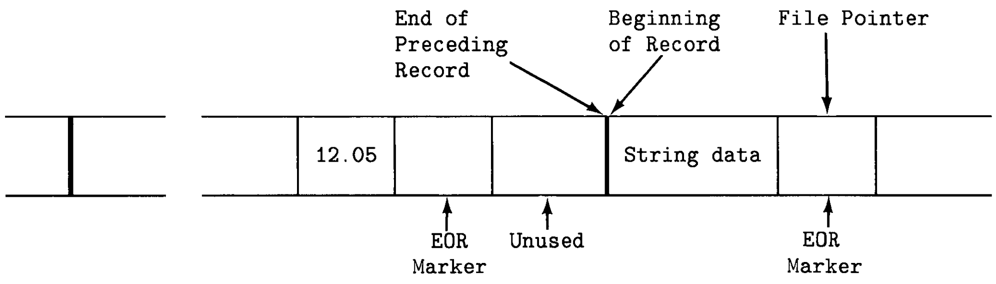


Figure 11-9. Record Markers

In the above example, there was enough space left in the current logical record to store the numeric item, so it was written. However, there was not enough space to store the string item (at least 4 bytes is required), so an EOR marker was written into the record. The file pointer was then placed at the beginning of the next logical record, and the string item was written. The file pointer is left at the location following the string item. (The *only* situation in which an EOR is not written into the logical record is when there is *exactly* enough room for a numeric item at the end of the record.)

The pointer will continue to move sequentially through the file as shown in the preceding examples, unless moved in another manner. For instance, executing an ASSIGN# statement on the same buffer number moves to the file pointer to the beginning of the file.

```
ASSIGN# 1 TO "Income84"
```

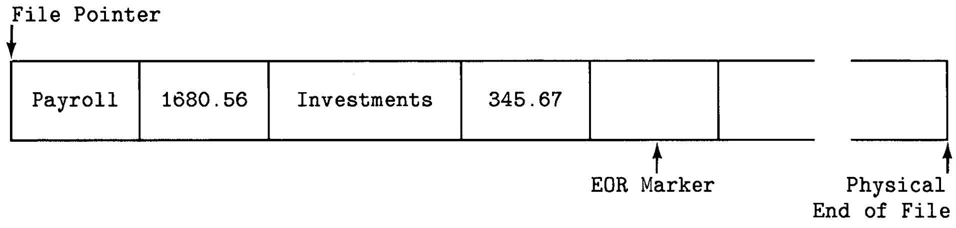


Figure 11-10. File Pointer Location 3

The movement of the file pointer and EOR marker influence the way in which the serial files are updated. For instance, if the pointer is reset to the beginning of the file (as in the preceding `ASSIGN#` statement) after serially reading a long list of data items, then a subsequent serial `PRINT#` statement will record new data items over the previous ones. In addition, an EOR marker is placed at the end of the new data items, so the result is that all previous data in the file is inaccessible.

```
PRINT# 1;"New data"
```

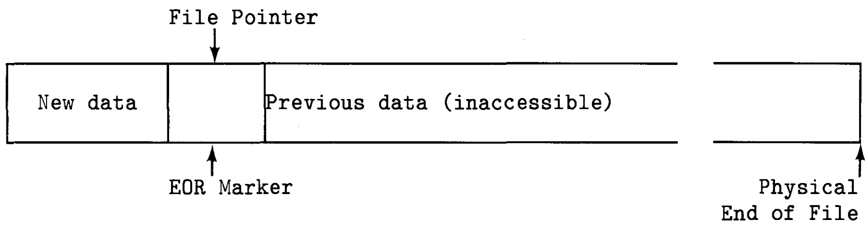


Figure 11-11. File Pointer Location 4

Extending Serial Files

These examples do not show that Technical BASIC files are *extensible*. That is, if you create a file and then attempt to serially write past the current *physical* end-of-file (not just past an EOR or EOF marker), then the system will automatically extend the file for you. Each extension is either one block or one logical record in length, whichever is *greater*.

Serial Read Operations

Data that has been stored in a data file must be retrieved (i.e., read back into computer memory) before it can be used by the program. Reading data from a file transfers a copy of the data through a buffer in computer memory.

When a file is opened, the file pointer is placed at the beginning of the file.

```
ASSIGN# 1 TO "Oct84Income"
```

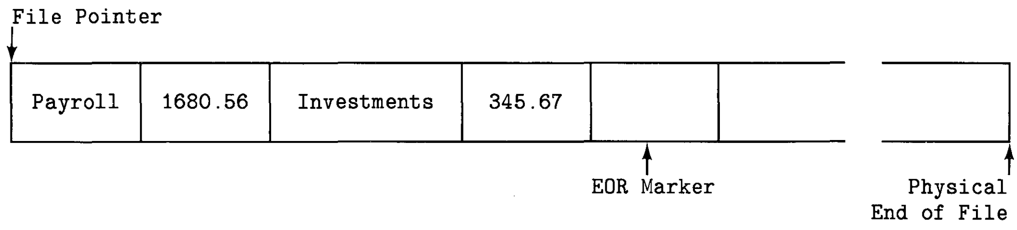


Figure 11-12. File Pointer Location 5

Serial reading is accomplished by the **READ#** statement; items in the data list are filled from left to right. As each data item is retrieved, the file pointer is updated to point to the next data item in the file. Items are accessed sequentially, *ignoring* any logical record boundaries.

```
READ# 1;IncNam$(1),TgtInc(1)
```

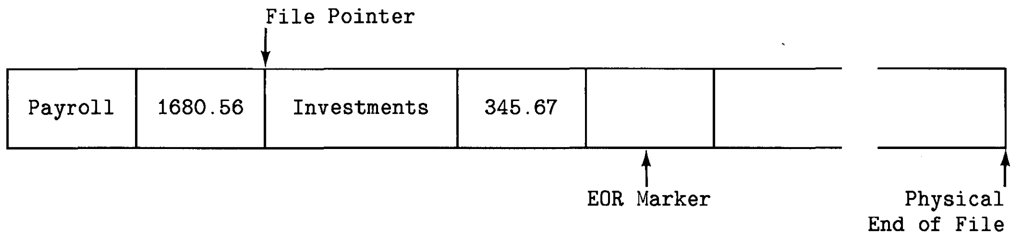


Figure 11-13. File Pointer Location 6

The variables used to read the data in the file must be of the same *general* data type as the data item (i.e., numeric or string), but they need not be of the same *specific* type (i.e., **INTEGER**, **SHORT**, or **REAL** for numeric items; or identical length for string items). However, matching specific types always works best because it prevents value range errors.

If a **READ#** statement attempts to read past an EOF marker, an error is reported. You can trap these errors with the **ON ERROR** statement. See the subsequent section called "Trapping EOF and EOR Conditions" for further details.

Both data stored serially and data stored randomly can be retrieved serially.

Random File Access

Random access allows you to move the file pointer to the beginning of any logical record within a file. This is in contrast to only setting the pointer to the beginning of a file for serial access, and then sequentially reading data items from the file and having the file pointer be updated automatically by the system. However, random access is like serial access after moving the pointer to the beginning of a logical record, because you will then serially access the data *in that record.*

Random Writing

Here is an example of creating a file with 12 logical records: each one contains target incomes (names and values) for a month of the year.

```
100  OPTION BASE 1 !  Lower bound of array subscripts.
110  DIM IncomeName$(2)
120  REAL TargetIncome(2)
130  !
140  IncomeName$(1)="Payroll"
150  IncomeName$(2)="Investments"
160  !
170  TargetIncome(1)=1680.56
180  TargetIncome(2)=345.67
190  !
200  ! Create and open a file.
210  CREATE "TgtInc84",12,42
220  ASSIGN# 1 TO "TgtInc84"
230  !
240  FOR Month=1 TO 12
250    PRINT# 1,Month !  Move pointer to start of record (random "seek").
260    FOR Category=1 TO 2
270      PRINT# 1;IncomeName$(Category),TargetIncome(Category) ! Serial wrt.
280    NEXT Category
290  NEXT Month
300  !
310  END
```

Here is a conceptual drawing of what is in *each* logical record.

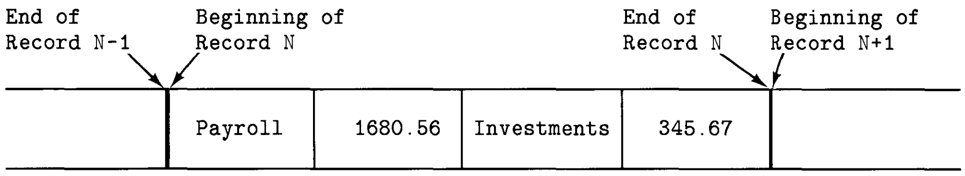


Figure 11-14. Logical Record

Here are the differences between serially and randomly writing to files.

- In order to randomly write to a data file, you must use a PRINT# statement that specifies a record number.

  ```
  200 PRINT# 1,3;Str$,Intgr  ! Write 2 data items in record 3.
  ```

  When a random PRINT# statement is executed, the file pointer is moved to the beginning of the specified record. The data items in the PRINT# statement are then recorded in the record, and an end-of-record (EOR) marker is placed after the last item (if there is at least 1 byte left in the record).

- If you want to merely position the file pointer at the beginning of a record, without writing any data, then execute a PRINT# statement specifying only the record number (omitting the data list).

  ```
  PRINT# 1,5
  ```

- Record divisions are **not** ignored, as they were in serial access. Thus, if you attempt to store more data in one logical record than that record will hold, an EOR error is reported:

  ```
  ERROR 69 : RANDOM OVF
  ```

  or

  ```
  ERROR 72 : RECORD
  ```

Randomly Reading

Here is an example of reading the data that was stored using random access methods. Note that the logical records are accessed in reverse order (12, 11, 10, ..., 1).

```
100  OPTION BASE 1
110  DIM IncomeName$(2)
120  REAL TargetIncome(2)
130  !
140  ! Open the file.
150  ASSIGN# 7 TO "TgtInc84" !  Buffer # 7.
160  !
170  FOR Month=12 TO 1 STEP -1 !  Access records in reverse order.
180    READ# 7,Month !  Move pointer to start of record (random "seek").
190    DISP "Month:";Month
200    DISP "---------"
210    FOR Category=1 TO 2
220      READ# 7;IncomeName$(Category),TargetIncome(Category) ! Serial read.
230      DISP "Income name:",IncomeName$(Category)
240      DISP "Target income:",TargetIncome(Category)
250      DISP
260    NEXT Category
270  NEXT Month
280  !
290  END
```

Here are the results of running the program.

```
Month: 12
---------
Income name:            Payroll
Target income:           1680.56

Income name:            Investments
Target income:           345.67

Month: 11
---------
Income name:            Payroll
Target income:           1680.56

Income name:            Investments
Target income:           345.67

.
.
.

Month: 1
---------
Income name:            Payroll
Target income:           1680.56

Income name:            Investments
Target income:           345.67
```

Randomly reading files is slightly different from serially reading files.

- In order to read data from a "random" record of a data file, you must use a **READ#** statement that specifies a record number.

  ```
  200 READ# 1,3;A$,I !        Read 2 data items from record 3.
  ```

  When a record is specified, the file pointer is moved to the beginning of that record. The data item(s) in the **READ#** statement are then transferred serially (through the buffer) into the specified variable(s).

- Logical record boundaries are **not** ignored. If you attempt to read more data items than are in the record, an EOR error will be reported (`ERROR 72 : RECORD`).

- If you want to merely position the file pointer at the beginning of a record without reading any data, then execute a **READ#** statement specifying only the record number (omitting the data list).

  ```
  READ# 1,3
  ```

As with serial reading, the variables used to read the data in the file must be of the same *general* data type as the data item (i.e., numeric or string), but they need not be of the same *specific* type (i.e., `INTEGER`, `SHORT`, or `REAL` for numeric items; or identical length for string items). However, matching specific data types is always best, because it eliminates the potential for value range errors.

Determining Data Types

A preceding section mentioned that each item written in a data file is preceded by a type field. You can use the `TYP` function to read this field and thereby determine the item's data type.

This function allows you to avoid errors such as attempting to read a string data item into a numeric variable. It also allows you to determine whether the file pointer is pointing at the current end-of-file (EOF) or end-of-record (EOR) marker.

Data-Type Field Values

Here is an example of using the `TYP` function:

```
ItemType=TYP(1)
```

The function reads the type field of the item at which the file pointer is currently pointing. The parameter passed to the function specifies which buffer is to be read. The preceding statement determines the type of the item at the current location of the file pointer in buffer number 1. An example program is shown below.

Here is the range of integer values that the `TYP` function can return, and the corresponding data types.

Table 11-3 Data Types

TYP Value	Data Type
1	Numeric
2	Full string
3	End-of-file marker
4	End-of-record marker
8	Start of string
9	Middle of string
10	End of string

Sensing EOF and EOR Conditions

Here is a simple example of using the TYP function to determine whether the file pointer is currently pointing at an EOF marker.

```
100  DEF FNEOF(BuffNo) = TYP(BuffNo)=3
```

Here is an example of using the function:

```
200 WhileNotEOF: IF NOT FNEOF(2) THEN ReadItem ELSE EndOfFile
```

Sensing an EOR marker is almost identical.

```
110  DEF FNEOR(BuffNo) = TYP(BuffNo)=4
```

Here is an example of using the function:

```
200  IF NOT FNEOR(2) THEN ReadItem ELSE NextRecord
```

Trapping EOF and EOR Conditions

There are certain conditions that you can encounter while writing and reading files that will generate an error. This section describes them.

The following operations will generate an end-of-file (EOF) or end-of-record (EOR) error condition:

- Attempting to read past either an EOF marker or the physical end of file (`ERROR 71: EOF`).
- Attempting to read more data items than there are in a logical record during a *random* read operation (`ERROR 72: RECORD`).
- Attempting to write more data than will fit in a logical record during a *random*[1] write operation (`ERROR 69: RANDOM OVF`).

[1] This error is only reported during random writes, because attempting to write past the physical end of file during a *serial* write causes the system to *automatically extend* the file.

Here is an example of using the ON ERROR mechanism to trap EOF errors while reading a file. The file is assumed to contain only string data.

```
100  DIM StringData$[65530]
110  !
120  Ask: DISP "Enter file name." @ INPUT File$
130       DISP "Is this correct?  '"&File$&"'  (Y/N)" @ INPUT Ans$
140       IF UPC$(Ans$[1,1])<>"Y" THEN Ask
150       !
160  ASSIGN# 2 TO File$ !  Open specified file.
170  !
180  ON ERROR GOTO ErrorTrap !  Set up branch for errors.
190  !
200  ! Loop until EOF (or other error).
210 NextItem: READ# 2;StringData$ !  Read as string; if error,
220                               !  branch to ErrorTrap.
230          DISP StringData$
240        GOTO NextItem
250        !
260 ErrorTrap: IF ERRN=71 THEN DISP "End of file found." @ GOTO Ask
270            ! ELSE ERRN<>71, so display error message.
280            ERRM
290  END
```

The program runs until either an EOF error (71) or another error is encountered. When an EOF is encountered, the message End of file found. is displayed, and the program asks for another file name. When another error is encountered, the system's normal error message is displayed. You can easily expand the ErrorTrap routine to respond to other file-related errors.

Using text/data Files

This section briefly describes how to write and read files of type `text/data`. This type of file provides a method of interchanging data files between Technical BASIC and the HP-UX system (they are HP-UX "ASCII" files).

Accessing this type of file with binary programs is described in the "Examples of File I/O" section in the "Binary Programs" chapter.

Writing to a text/data File

This program shows how to open and write data into a file of type `text/data`. If the file does not already exist, the BASIC system will create it for you. Numeric and string data items are then written into the file.

```
100 INTEGER IntVar
110 IntVar=32000
120 !
130 SHORT ShortVar
140 ShortVar=3e+031
150 !
160 REAL RealVar
170 RealVar=1e+308
180 !
190 DIM StringVar$[20]
200 StringVar$="This is a string."
210 !
220 ASSIGN 14 TO "text_file" !       Assign a file selector.
230 !
240 OUTPUT 14 ; IntVar;ShortVar !    Write two values into file.
250 OUTPUT 14 ; RealVar;StringVar$ ! Write two more values into file.
260 !
270 ASSIGN 14 TO "*" !               Close file.
280 !
290 END
```

Note that the items specified in the `OUTPUT` statement are written according to the rules of the `OUTPUT` statement; see the *HP-UX Technical BASIC I/O Programming Guide* or the *HP-UX Technical BASIC Language Reference* for details.

In this example, the items in the `OUTPUT` statements are separated by semicolons; therefore, the items will **not** be separated (in the output data stream) by an end-of-line (EOL) sequence, which is normally a carriage-return followed by a line-feed (control characters). However, the EOL sequence is automatically sent after the last item in the `OUTPUT` statement (unless suppressed with a semicolon or comma).

Note also that the OUTPUT statement does not put end-of-record (EOR) or end-of-file (EOF) markers in the file.

Reading from a text/data File

This example reads the data from the `text/data` file written with the preceding example. It uses this general rule: *read the file in the same way that it was written.*

```
100 INTEGER IntVar
110 IntVar=-1
120 !
130 SHORT ShortVar
140 ShortVar=-1
150 !
160 REAL RealVar
170 RealVar=-1
180 !
190 DIM StringVar$[20]
200 StringVar$="Initial value."
210 !
220 DISP "Value of IntVar     = ";IntVar ! Show values BEFORE reading file.
230 DISP "Value of ShortVar   = ";ShortVar
240 DISP "Value of RealVar    = ";RealVar
250 DISP "Value of StringVar$ = ";StringVar$
260 DISP
270 !
280 ASSIGN 14 TO "text_file" !      Assign a file selector.
290 !
300 ENTER 14 ; IntVar,ShortVar !    Read two values from file.
310 ENTER 14 ; RealVar,StringVar$ ! Read two more values from file.
320 !
330 DISP "Value of IntVar     = ";IntVar ! Now show values read FROM FILE.
340 DISP "Value of ShortVar   = ";ShortVar
350 DISP "Value of RealVar    = ";RealVar
360 DISP "Value of StringVar$ = ";StringVar$
370 !
380 ASSIGN 14 TO "*" !              Close file.
390 !
400 END
```

Here is the output of the program.

```
Value of IntVar     = -1
Value of ShortVar   = -1
Value of RealVar    = -1
Value of StringVar$ = Initial value.

Value of IntVar     =  32000
Value of ShortVar   =  3e+031
Value of RealVar    =  1e+308
Value of StringVar$ =  This is a string.
```

C Binaries 12

Introduction

Generally, you will be using the Technical BASIC system to execute programs written in the Technical BASIC language. However, you can also write programs in the C language and then call (execute) them from Technical BASIC. In this manual, such programs are termed binary programs. The term "binary" was probably coined because the programs written in another language and compiled into executable object code cannot be easily read by humans from the BASIC system—they look like just a bunch of binary patterns.

Binary programs are useful in the following situations:

- An application is already written in C, and you don't want or have time to translate it into Technical BASIC code.
- C supports a feature that is not available in Technical BASIC, or the C version runs faster that the BASIC version.

Chapter Contents

Tasks/Topics	Page
Overview	12-2
Structure of C programs and binaries	12-2
Compiling and linking C binaries	12-3
Summary of parameter-type matching	12-5
Restrictions	12-5
A closer look at compiling and linking	12-7
Examples of passing parameters	12-9
Examples of file I/O	12-19

Overview

This section briefly covers the following topics. It is intended to quickly give you a global view of using C binaries. Specific details of each topic are presented in the subsequent sections.

- Structure of C programs and binaries.
- Procedure for creating C binaries.
- Parameter-type matching.
- Restrictions on C binaries.

Structure of C Programs and Binaries

C programs have the following structure. The unshaded portion can be used up as a binary—that is, it can be separately compiled and linked to the BASIC system, and then called as a separate entry point by a BASIC program.

```
/* C 'program' that calls a C 'function'. */
main()
{
  int pass_param;

  pass_param = 7;
  printf("Before calling 'entry_pt':\n");
  printf("pass_param= %2d \n\n",pass_param);

  entry_pt(&pass_param);

  printf("After  calling 'entry_pt':\n");
  printf("pass_param= %2d \n\n",pass_param);

} /* End of main. */


/* Now the C function. */
entry_pt(formal_param)

  int *formal_param;

{
   /* Double the value passed to the routine. */
   *formal_param = *formal_param * 2;
}
```

C Program

C Function[1] (can be used as a binary)

[1] Note that this is not the same as a BASIC function in that it does not return a value like a BASIC function (such as **SIN(X)** or **FNmyfunc(Argument)**).

Here are the results of running the program:

```
Before calling 'entry_pt':
pass_param=  7

After  calling 'entry_pt':
pass_param= 14
```

Note that the C function[1] named `entry_pt` was able to modify the variable `pass_param`'s contents, since the parameter was passed by reference.

Compiling and Linking

1. If the C code is structured as a program, you will need to re-structure it so that it is a stand-alone C function. Store this code in a file separate from the original copy. In the above example, this is the routine named `entry_pt`.

   ```
   entry_pt(formal_param)

     int *formal_param;

   {
      /* Double the value passed to the routine. */
      *formal_param = *formal_param * 2·
   }
   ```

   NOTE

   You should have, at this point, thoroughly debugged the C binary by calling it from a C program (as in the example C program above). The reason for this approach is that even though BASIC attempts to trap errors, it cannot trap them all.

2. While in the HP-UX shell, compile the C binary and link it to any libraries required to resolve the external references. A shell script is provided for this purpose. When you use the script, specify the name of the file in which you want the binary to be stored. For instance, if the binary is in a file named `bin1.c`, then the script call would be:

   ```
   $ /usr/bin/makebin_c  bin1 Return
   ```

[1] Note that this is not the same as a BASIC function in that it does not return a value like a BASIC function (such as **SIN(X)** or **FNmyfunc(Argument)**).

3. Enter the Technical BASIC system, and type in and run a BASIC program that loads and calls the binary. Make sure that the parameters passed to the binary match those expected by the subsequent table for a complete list of correspondence between BASIC and C parameters.)

   Here is an example BASIC program that calls the preceding binary:

   ```
   100  LOADBIN "bin1"
   110  !
   120  INTEGER WholeNumber
   130  WholeNumber=7
   140  !
   150  CLEAR
   160  DISP "Before CALLBIN:"
   170  DISP "WholeNumber =";WholeNumber
   180  DISP
   190  CALLBIN "entry_pt" (WholeNumber)
   200  DISP
   210  DISP "After CALLBIN:"
   220  DISP "WholeNumber =";WholeNumber
   230  !
   240  SCRATCHBIN "bin1"
   250  END
   ```

   The `LOADBIN` statement (line 100) links the binary to BASIC. This example assumes that the binary is the file named `bin1` in the current working directory. (If it is not in a file in that directory, then you would need to specify a path name.)

   The BASIC program next assigns a value to an `INTEGER` variable (line 130), and then displays the value (lines 160 and 170).

   The `CALLBIN` statement (line 190) branches to the specified entry point in the binary; in this case, the entry point is named `entry_pt`.

   After the binary has finished execution, it returns control to the BASIC program. In this example, the BASIC program displays the modified value of the variable `WholeNumber`. Note that the BASIC variable `WholeNumber` is passed *by reference*, which allows the binary to modify the variable's contents. Passing parameters is further described in subsequent sections.

   Once you no longer need the binary, you can unlink it from BASIC with the `SCRATCHBIN` statement (line 240). Note that the file is still in the HP-UX file system; however, it is not linked to Technical BASIC any longer, and is therefore inaccessible from BASIC.

Summary of Parameter-Type Matching

Here is a list of all the pass parameter types that you can send to a C binary, along with the corresponding C formal parameter types.

Table 12-1. Matching Parameters in BASIC Programs and C Binaries

BASIC Data Type	C Data Type
simple `INTEGER`	`int`
`INTEGER` array	array of `int`
simple `SHORT`	`float`
`SHORT` array	array of `float`
simple `REAL`	`double`
`REAL` array	array of `double`
simple string	`char`, or array of `char`[1]
string array	array of `char`

Restrictions

You can use almost all the features of a language in a binary program. However, there are a few things that you cannot do with C binaries.

Maximum Number of Binaries

At any one time, there can be up to 5 C binaries loaded (using `LOADBIN`). If there are Pascal, Fortran, and C binaries concurrently loaded, then the maximum number of C binaries is 3.

Note, however, that an individual binary may contain *several* entry points—C functions—as long as they are placed in one object file.

File I/O Restrictions

Binary programs can perform I/O operations on files, with only one restriction: if the binary is to access a file of `text/data` (HP-UX "ASCII"), then the binary must open the file, access the information, and close the file while BASIC is not accessing it. For instance, if a BASIC program has a particular `text/data` file currently open, then no binary should access that file. After BASIC has closed it, the binary may open it, perform I/O operations on it, and then close it. (The converse situation has the same restriction.)

[1] Simple BASIC strings can be passed either by reference or by value. However, if a string is passed by value, then it must be declared as an array of type `char` in the C binary.

Note, however, that there is no additional restriction on the number of files that BASIC or C may have open at one time. For instance, BASIC can still have up to 10 files of type `text/data` open simultaneously, while C can have up to 20 files open at one time.

Examples of C file I/O are given in the last section of this chapter.

HP-UX Environment May Not Be Accessible

HP-UX environment variables, such as `TERM` and `PATH`, may not be accessible to binaries System calls, such as `ioctl`, should be used with caution in binaries.

Standard I/O Streams Are Not Accessible

Binaries should **not** perform operations on the "standard I/O streams"—namely, displaying on the screen (`stdout` and `stderr` in C) or getting characters from the keyboard (`stdin` in C). For example, there is no guarantee that the C standard I/O library function `printf` will work in all binaries, although you may get it to work in some instances.

Error Trapping in Binaries

Technical BASIC usually recovers gracefully from errors encountered while using binaries. However, there are some errors that BASIC cannot handle, and thus may even cause the HP-UX system to log you out. Here are some suggestions on how to avoid this type of situation.

1. The best approach to avoiding errors when using binaries is to **thoroughly** test the binary (with a stand-alone program) **before** calling it from BASIC.
2. When you are ready to actually call it from BASIC, make sure that the you match pass parameters correctly: same number of parameters, correct type-matching, and in the right order. (Later sections provide additional examples of passing parameters to C-language binaries.) Also be sure that the binary stays within the bounds of data structures and arrays, especially when the binary uses pointers.
3. If you do get an error while loading or executing a binary, you should fix the problem in the binary first. Then when back in BASIC, you should use `SCRATCHBIN` to unload the old copy before using `LOADBIN` to load the latest copy. This action must be taken, because `LOADBIN` will not load a binary if a binary of the same name is currently loaded.

A Closer Look at Compiling and Linking

The example binary used in this section is the same one used earlier in the chapter; it doubles the value of an integer that is passed to it.

```
entry_pt(formal_param)

int  *formal_param;

{
   /* Double the value passed to the routine. */
   *formal_param = *formal_param * 2;
}
```

While in the HP-UX shell, compile the C binary and link it to any libraries required to resolve the external references. Use the `makebin_c` shell script for this purpose; the default location of this script is in the `/usr/bin` directory. Assuming the C program is called `bin1.c`, you could type:

```
$  /usr/bin/makebin_c  bin1  Return
```

Here are the contents of the `makebin_c` shell script (for Series 500 systems):

```
for filec
do
   cc  -c  $filec.c
   ld  -rd  -o $filec  $filec.o  /usr/lib/bcrt0.o  -lc
done
```

`filec`	is a variable that contains the name of the file specified when the script was executed; in this example, this variable contains `bin1`.
`cc`	is the "C compiler" command.
`-c`	specifies not to generate the normal linked, executable (`a.out`) object file. Instead, the C compiler is to generate an unlinked, relocatable object file—a file named `bin1.o` in this case.
`ld`	is the "link editor" command.
`-rd`	options indicate (respectively): the specified object file (`bin1.o`) is to be loaded as relocatable (re-linkable) code; the definition of "COMMON" storage is to be forced.
`-o`	option specifies that the object file is to be named `bin1`, rather than given the default name `a.out`.
`/usr/lib/bcrt0.o`	is a special version of the `crt.o` library that must be searched when linking C binaries to BASIC.
`-lc`	option specifies that the C libraries are to be made accessible to the binary. (With Series 200/300 systems, the `makebin_c` shell script specifies `-lb` to search the `libb.a` library instead of the `libc.a` library.)

If there are still unresolved references[1] after executing this script, then you will have to specify additional libraries:

- for system libraries (`/lib/libxxx.a`), include the appropriate `ld` option—such as `-lm`;
- for other libraries, specify the library by name—such as the `/usr/lib/bcrt0.o` library file specified in this script.

Now you are ready to enter the Technical BASIC system, load the binary, and then call it from BASIC.

[1] If the **LOADBIN** or **CALLBIN** statements report an error, there are probably unresolved references in the binary. You can use the HP-UX **nm** command to find them.

Example C Binaries

This section contains examples and corresponding explanations of the following tasks:

- Passing parameters between BASIC and C binaries.
    - Simple numerics.
    - Numeric arrays.
    - Simple strings.
    - String arrays.
- Using `text/data` type files for data interchange between BASIC and C binaries.

If you have trouble understanding the mechanisms of "passing by reference" or "passing by value", then you may want to study further examples of passing parameters in the "Subprograms" section of the "User-Defined Functions and Subprograms" chapter. Additional details of `text/data` files are provided in the "Data Storage and Retrieval" chapter section called "Using text/data Files".

Passing Simple Numeric Parameters

There are three BASIC data types: `INTEGER`, `SHORT`, and `REAL`. You can pass all of these types to C programs. Here is the required correspondence between BASIC simple-numeric pass parameters and C formal parameters:

Table 12-2. Matching Simple Numeric Parameters

BASIC Pass Parameter	Corresponding C Formal Parameter
`INTEGER`	`int`
`SHORT`	`float`
`REAL`	`double`

Here are general rules for passing these types of parameters (examples are given subsequently):

- If a parameter is to be passed **by reference**, then it must be placed in a BASIC variable. The C binary must declare the corresponding parameter as a pointer (to the appropriate C type listed above).
- If a parameter is to be passed **by value**, the BASIC parameter must be an expression (such as a numeric literal, a variable enclosed in parentheses, or a combination of numeric items containing numeric operators). The corresponding C formal parameter must be declared as a variable of the appropriate type—not as a pointer.

This BASIC program passes three parameters to the subsequent C binary.

```
100 INTEGER RadiusB
110 RadiusB=10
120 REAL AreaB
130 !
140 LOADBIN "area"
150 CALLBIN "area" (PI,(RadiusB),AreaB)
160 DISP "Area of circle with radius";RadiusB;"=";AreaB
170 SCRATCHBIN "area"
180 END
```

Here is a C binary that would work with the above BASIC program.

```
area(Pi,RadiusC,AreaC)

  double  Pi;
  int     RadiusC;
  double  *AreaC;

{
  *AreaC = Pi * RadiusC * RadiusC;
}
```

Here are the results of running the program:

```
Area of circle with radius 10 is 314.159265358979
```

The first BASIC pass parameter, `PI`, is passed by value since it is a numeric function (which qualifies it as a numeric expression). The corresponding C formal parameter is of type `double`, since `PI` is a function of BASIC type `REAL`.

The second BASIC pass parameter, `(RadiusB)`, is also passed by value since it has been enclosed in parentheses (which makes it an expression). The corresponding C formal parameter is of type `int`. Note that this parameter must **not** be passed by reference, since C expects a value, not an address (a C pointer).

The third BASIC pass parameter, `Area`, is passed by reference since it is a variable which is not part of an expression. The corresponding C formal parameter is of type *pointer to* `double`, as indicated by the leading `*`. A pointer variable is one that contains the *address* of the variable, rather than its *value*. This is required because the corresponding BASIC pass parameter is passed "by address" (by reference). Passing a variable by reference allows the binary to modify the BASIC variable's contents, thereby allowing information to be "passed back" to BASIC.

Passing Numeric Array Parameters

Here is the required correspondence between BASIC numeric-array pass parameters and C formal parameters:

Table 12-3. Matching Numeric Array Parameters

BASIC Pass Parameter	Corresponding C Formal Parameter
`INTEGER` array	Array of `int`
`SHORT` array	Array of `float`
`REAL` array	Array of `double`

Note that arrays are **always passed by reference** to binaries.

The following BASIC program, a modification of the preceding example, passes two numeric arrays to a C binary.

```
100 INTEGER RadiiB(4) ! 5 elements (OPTION BASE 0).
110 FOR I=0 TO 4
120   RadiiB(I)=I
130 NEXT I
140 !
150 REAL AreasB(4)
160 !
170 LOADBIN "arrays"
180 CALLBIN "arrays" (PI,RadiiB(),AreasB())
190 DISP "Radii  Areas"
200 DISP "-----  -----"
210 FOR I=0 TO 4
220   DISP USING "DD.DD,XX,DD.DD" ; RadiiB(I),AreasB(I)
230 NEXT I
240 SCRATCHBIN "arrays"
250 END
```

Here is a C binary that would work with the preceding BASIC program.

```
arrays(Pi,RadiiC,AreasC)

  double  Pi;
  int     *RadiiC;   /* Pointer to array.     */
  double  AreasC[5]; /* Can also be AreasC[]  */

{
  int i;

  for (i=0; i<5; i++) /* Assume 5 elements in each array. */
    AreasC[i] = Pi * *(RadiiC+i) * *(RadiiC+i);
}
```

Here are the results of running the BASIC program:

```
Radii  Areas
-----  -----
 0.00   0.00
 1.00   3.14
 2.00  12.56
 3.00  28.27
 4.00  50.26
```

Notice that the binary assumes that the calling BASIC program will send an array with at least five elements. A more general method would be to pass arrays of variable sizes to the binary. Here are two possible methods:

- By passing parameter(s) that indicate the number of elements (and dimensions).
- By assigning a unique "flag" value to an array element to indicate that it is the last element in the array.

Note that the C array declarations in the program use different notation:

```
int     *RadiiC;    /* Pointer to array.     */
double  AreasC[5]; /* Can also be AreasC[]  */
```

These two declarations are *equivalent* in purpose because they each declare a pointer to the first element of an array (i.e., the element with subscript 0). The notation you use in the declaration dictates the notation that you will use in accessing array elements. For instance, individual elements of the `RadiiC` array are accessed by specifying the subscript: `Areas[i]`, for example. The elements of the `RadiiC` array are accessed by using pointer expressions: for instance, `*(RadiiC+i)`.

Passing Simple String Parameters

Here is the required correspondence between BASIC simple-string pass parameters and C formal parameters:

Table 12-4. Matching Simple String Parameters

BASIC Pass Parameter	Corresponding C Formal Parameter
Simple string	`char` or Array of `char`[1]

[1] Simple BASIC strings can be passed either by reference or by value. However, if a string is passed by value, then it must be treated as an array of type `char` in the C binary.

Here are general rules for passing these types of parameters (examples are given subsequently):

- If a parameter is to be passed **by reference**, then it must be placed in a BASIC variable. The C binary must declare the corresponding parameter as a pointer to the appropriate type listed above. Note that even though a C binary may change a BASIC string variable's contents, it cannot change the BASIC string variable's length. (This topic is discussed in the section called "BASIC String Length Headers".)
- If a parameter is to be passed **by value**, the BASIC parameter must be an expression (such as a quoted string literal, a variable enclosed in parentheses, or a combination of string items containing the string concatenation operator &). The corresponding C formal parameter must be declared as a variable of the appropriate type—not as a pointer.

The following BASIC program and C binary illustrate passing string parameters. (The string-length header does not require modification in this program; see the next program for example techniques.)

```
100 DIM ByRef$[9],ByValue$[11]
110 ByRef$="variable"&CHR$(0)
120 ByValue$="expression"&CHR$(0)
130 DISP
140 DISP "ByRef$ before call   = '";ByRef$;"'"
150 DISP "ByValue$ before call = '";ByValue$;"'"
160 DISP
170 LOADBIN "strings1"
180 CALLBIN "strings1" (ByRef$,ByValue$&"")
190 !
200 DISP "ByRef$ after call    = '";ByRef$;"'"
210 DISP "ByValue$ after call  = '";ByValue$;"'"
220 SCRATCHBIN "strings1"
230 END
```

Here is the corresponding C binary.

```
strings1(ByRef,ByValue)

char  *ByRef,
       ByValue[10];

{
  int   i;

  /* Assign new value to formal parameter 'ByRef'.    */
  strcpy(ByRef,"modified");

  /* Assign new value to formal parameter 'ByValue'. */
  for (i=0; ByValue[i]!='\0'; i++)
    ByValue[i] = 'x';

}
```

Here are the results of running the program.

```
ByRef$ before call   = 'variable'
ByValue$ before call = 'expression'

ByRef$ after call    = 'modified'
ByValue$ after call  = 'expression'
```

Note that the contents of variables passed by value to the binary are not changed; only the contents of variables passed by reference can be modified by binaries.

BASIC String Length Headers

BASIC strings have a length header that indicates how many characters the string *currently* contains. However, this header is not passed to any binary. C strings have no such header; they are instead terminated by the null control character: `\0` in C; `CHR$(0)` in BASIC. Thus, C binaries cannot modify the BASIC string variable's length.

For instance, suppose that you pass a string variable (by reference) to a C binary. The binary then proceeds to change the length of the string, but it *does not* modify the BASIC string's length header (because it has no access to it). Thus upon returning to BASIC, there is no indication that the length of the string variable is any different than when it was passed to the binary.

Here is an example that illustrates this behavior:

```
100 DIM ByRef$[10]
110 ByRef$="variable"
120 !
130 DISP "ByRef$ before call = '";ByRef$;"'"
140 DISP "String length = ";LEN(ByRef$)
150 DISP
160 CALLBIN "strings2" (ByRef$)
170 !
180 DISP "ByRef$ after call  = '";ByRef$;"'"
190 DISP "String length = ";LEN(ByRef$)
200 SCRATCHBIN "strings2"
210 END
```

Here is the corresponding C binary:

```
strings2(ByRef)

char  *ByRef;

{
  /* Now make string shorter. */
  strcpy(ByRef,"len=5");
}
```

Here is the program's output:

```
ByRef$ before call = 'variable'
String length =  8

ByRef$ after call  = 'len=5le'
String length =  8
```

The BASIC program sets the variable's length in the assign statement (line 110), and then displays its value and length.

The binary then assigns the string a new value. The new length of this string, according to C, is five characters. The binary then returns control to BASIC. Since the BASIC string variable was passed by reference (address), its *contents* are affected by the binary; however, the BASIC variable's *length* is *not* changed accordingly.

The BASIC program displays the string's contents and length. This display shows that only the first six characters of the variable were changed: the five characters `len=5`; and the null character, `\0`, which is not displayed unless the "display functions" mode is in effect. The BASIC variable's length and the remaining two characters, `le`, are not changed.

There are two steps in the general work-around for this type of situation:

1. Before passing the variable (by reference), pad the string with blank characters to the *maximum* length of string that the binary can return. For instance, the following statement pads the BASIC string variable with trailing blanks and sets its length to the maximum (dimensioned) length.

   ```
   ByRef$[LEN(ByRef$)+1]=" "
   ```

   Note that this particular statement will cause an error if the string length is already equal to the maximum (dimensioned) length.

2. After returning to the BASIC program, determine the string's new length.

   a. Search the returned string for a null character, CHR$(0), and then set the string length to 1 less than the position of the null.

      ```
      NullPos=POS(ByRef$,CHR$(0))
      ByRef$=ByRef$[1,NullPos-1]
      ```

   b. Pass a string length parameter (by reference) to the binary. After the binary changes the string's length, it can set the length parameter accordingly and then pass it back to BASIC.

      ```
      ByRef$=ByRef$[1,Length]
      ```

Passing String Arrays

Here is the required correspondence between BASIC string-array pass parameters and C formal parameters:

Table 12-5. Matching String Array Parameters

BASIC Pass Parameter	Corresponding C Formal Parameter
String array	Array of `char`

As with numeric arrays, **string arrays can only be passed by reference**. The C binary must declare the corresponding formal parameter appropriately. Here are some examples of string-array declarations and pass parameter lists, and their corresponding C declaractions.

BASIC Declaration & Call **C Declaration**

```
100  OPTION BASE 1                       char *StringArr;
110  DIM StrArray$(5)                         or
120  CALLBIN "entry_pt"(StrArray$())     char StringArr[5][18];
                                              or
                                         char StringArr[][18];


100  OPTION BASE 0                       char *StringArr;
110  DIM StrArray$(9)[30]                     or
120  CALLBIN "entry_pt"(StrArray$())     char StringArr[10][30];
                                              or
                                         char StringArr[][30];

100  OPTION BASE 1                       char *StringArr;
110  DIM StrArray$(5,10)[20]                  or
120  CALLBIN "entry_pt"(StrArray$(,))    char StringArr[5][10][20];
                                              or
                                         char StringArr[][10][20];


100  OPTION BASE 0                       char *StringArr;
110  DIM StrArray$(5,10)[50]                  or
120  CALLBIN "entry_pt"(StrArray$(,))    char StringArr[6][11][50];
                                              or
                                         char StringArr[][11][50];
```

The implications of using the declaration `char *StringArr` versus `char StringArr[]` are the same as for numeric array declarations: the notation you use in the declaration dictates the notation that you must use to specify individual array elements.

Here is an example of how to specify array elements when the first declaration method has been used (that is, *StringArr):

```
str_array1(st_arr)

char  *st_arr;

{
  strcpy(st_arr,"Line a");
  strcpy(st_arr+18,"Line b");
  strcpy(st_arr+36,"Line c");
  strcpy(st_arr+54,"Line d");
  strcpy(st_arr+72,"Line e");
}
```

Here is an example of specifying array elements when either the second or the third declaration method has been used (that is, `char StringArr[5][10][20];` or `char StringArr[][10][20];`):

```
str_array2(st_arr)

char  st_arr[5][18];

{
  strcpy(st_arr[0],"Line a");
  strcpy(st_arr[1],"Line b");
  strcpy(st_arr[2],"Line c");
  strcpy(st_arr[3],"Line d");
  strcpy(st_arr[4],"Line e");
}
```

The implications of using `StringArr[5]` versus `StringArr[]` are that the former specifies the (maximum) size of the array, while the latter allows the size of the array to vary.

Examples of C File I/O

C binaries can perform general file I/O operations—as long as they open files, access them, and close files while BASIC is not currently accessing it. For instance, if a BASIC program currently has a particular `text/data` file open (with `ASSIGN`), then a binary should **not** access that file. However, once BASIC closes the file, the binary may access it. (The converse situation has the same restrictions.)

Note, however, that there is no additional restriction on the number of files that BASIC or C may have open at one time. For instance, BASIC can still have up to 10 files of type `text/data` open simultaneously, while C can have up to 20 files open at one time.

In addition, if BASIC is also to use a file used by a binary, then the file type must be of the type which which BASIC calls type `text/data`; in HP-UX, these files are known as "ASCII" files.

- If a file already exists, you can determine whether or not it is a `text/data` file by using the CAT statement.
- If you want to create it with BASIC, use the `ASSIGN` statement. See the "Using text/data Files" section of the "Data Storage and Retrieval" chapter for an example of creating and using this type of file.

This section shows examples of BASIC programs which call C binaries that write to a file and read from it.

BASIC Calls a Binary that Writes to a File

The first program passes a file name, a string array, and a parameter indicating the number of array elements to a binary; the binary then writes the data into the specified file. (Note that in this particular example the BASIC program does not create, write to, or read from the file—although it certainly could. See the "Using text/data Files" in the "Data Storage and Retrieval" chapter for examples.)

```
100 FileName$="sometext"
110 !
120 DIM StringArray$(10)[79]
130 StringArray$(0)="This is the first  line of text."&CHR$(0)
140 StringArray$(1)="This is the second line of text."&CHR$(0)
150 StringArray$(2)="This is the third  line of text."&CHR$(0)
160 StringArray$(3)="This is the fourth line of text."&CHR$(0)
170 StringArray$(4)="This is the fifth  line of text."&CHR$(0)
180 !
190 LOADBIN "text_write"
200 CALLBIN "text_write" (FileName$,StringArray$(),5)
210 DISP "Finished writing to file."
220 !
230 SCRATCHBIN "text_write"
240 END
```

Here is a listing of the binary that writes the string array into the specified file.

```
#include <stdio.h>

text_write(file_name,str_array,nlines)

char  *file_name,
      str_array[][79];
int   nlines;

{

  FILE  *file_pointer,
        *fopen(),
        *fclose();
  int   line;

  /* Open the file for writing.        */
  /* (File must NOT be open in BASIC.) */
  file_pointer = fopen(file_name,"w");

  /* Write the string array into the file. */
  for (line = 0; line < nlines; line++)
    fprintf(file_pointer,"%s \n",str_array[line]);

  /* Close the file before returning to BASIC. */
  fclose(file_pointer);

}
```

BASIC Calls a Binary that Reads the File

Here is a BASIC program that calls another binary which reads the data written by the preceding BASIC and binary programs.

```
100 FileName$="sometext"
110 !
120 DIM StringArray$(10)[79]
130 FOR Line=0 TO 4 !     Fill strings with spaces
140                 !     (to set string length)
150   StringArray$(Line)[2]=" "
160 NEXT Line
170 !
180 LOADBIN "text_read"
190 CALLBIN "text_read" (FileName$,StringArray$(),5)
200 !
210 FOR Line=0 TO 4
220   DISP StringArray$(Line)
230 NEXT Line
240 SCRATCHBIN "text_read"
250 END
```

Here is a binary that reads the file.

```
#include <stdio.h>

text_read(file_name,str_array,nlines)

char  *file_name,
      str_array[][79];
int   nlines;

{

  FILE  *file_pointer,
        *fopen(),
        *fclose();
  int   line,
        i;
  char  c;

  /* Open the file for reading.        */
  file_pointer = fopen(file_name,"r");

  if(file_pointer != NULL)
  /* Then file was opened w/o errors. */
  {
    /* Read the data in the file line by line. */
    for (line = 0;  line < nlines; line++)
    {
      i = 0; /* copy line char-by-char */
      while ((c = getc(file_pointer)) != '\n')
        str_array[line][i++] = c;
    } /* end for */

  } /* end if */
  else
  /* File was not opened, or other error occurred. */
    strcpy(str_array[0],"ERROR");

  /* Close the file before returning to BASIC. */
  fclose(file_pointer);

}
```

Pascal Binaries 13

Introduction

Generally, you will be using the Technical BASIC system to execute programs written in the Technical BASIC language. However, you can also write programs in the Pascal language and then call (execute) them from Technical BASIC. In this manual, such programs are termed binary programs. The term "binary" was probably coined because the programs written in another language and compiled into executable object code cannot be easily read by humans from the BASIC system—they look like just a bunch of binary patterns.

Binary programs are useful in the following situations:

- An application is already written in Pascal, and you don't want or have time to translate it into Technical BASIC code.
- Pascal supports a feature that is not available in Technical BASIC, or the Pascal version runs faster that the BASIC version.

Chapter Contents

Tasks/Topics	Page
Overview	13-2
Structure of Pascal programs and binaries	13-2
Compiling and linking Pascal binaries	13-3
Summary of parameter type matching	13-5
Restrictions	13-6
A closer look at compiling and linking	13-8
Examples of passing parameters	13-10
Examples of file I/O	13-22

Overview

This section briefly covers the following topics. It is intended to quickly give you a global view of using Pascal binaries. Specific details of each topic are presented in the subsequent sections.

- Structure of Pascal programs and binaries.
- Procedure for creating Pascal binaries.
- Parameter-type matching.
- Restrictions on Pascal binaries.

Structure of Pascal Programs and Binaries

Here is an HP Pascal program that contains (and uses) a module. The unshaded portion (the module) can be used up as a binary—that is, it can be separately compiled and linked to the BASIC system, and then called as a separate entry point by a BASIC program.

```
program callbin1(output);

module pmod1;

export
  procedure entpt(var formal_param : integer);

implement
  procedure entpt(var formal_param : integer);
    begin
      formal_param := formal_param * 2;
    end;

end;

import pmod1;

var
  pass_param : integer;

begin

  pass_param := 7;
  writeln('Before calling "entpt":');
  writeln('pass_param=',pass_param);

  entpt(pass_param);

  writeln('After  calling "entpt":');
  writeln('pass_param=',pass_param);

end.
```

Pascal Module (can be used as a binary)

Pascal Program

Here are the results of running the program:

```
Before calling "entpt":
pass_param=  7

After  calling "entpt":
pass_param= 14
```

Note that the Pascal procedure named **entpt** was able to modify the variable **pass_param**'s contents, since the parameter was passed by reference.

Compiling and Linking

1. If the Pascal code is structured as a program, you will need to re-structure it as a module that exports the desired entry points. Store this code in a file separate from the original copy. In the above example, this is the **module** named **pmod1**. (Notice that the semicolon following module's the **end** statement has been changed to a period to allow separate compilation.)

```
module pmod1;

export
  procedure entpt(var formal_param : integer);

implement
  procedure entpt(var formal_param : integer);
    begin
      formal_param := formal_param * 2;
    end;

end.
```

NOTE

You should have, at this point, thoroughly debugged the Pascal module by calling it from a Pascal program (as in the example Pascal program above). The reason for this approach is that even though BASIC attempts to trap errors, it cannot trap them all.

2. While in the HP-UX shell, compile the Pascal binary and link it to any libraries required to resolve the external references. A shell script is provided for this purpose. When you use the script, specify the name of the file in which you want the binary to be stored. For instance, if the binary is in a file named **pmod1.p**, then the script call would be:

```
$ /usr/bin/makebin_p  pmod1 [Return]
```

3. Enter the Technical BASIC system, and type in and run a BASIC program that loads and calls the binary. Make sure that the parameters passed to the binary match those expected by the binary—in both number and in type of each parameter. (See the subsequent table for a complete list of correspondence between BASIC and Pascal parameters.)

   Here is an example BASIC program that calls the preceding binary:

```
100 INTEGER PassParam,echo !  Declare simple numeric types.
110 PassParam=7
120 !
130 DISP " Before binary called. "
140 DISP "PassParam = ";PassParam
150 DISP
160 !
170 LOADBIN "pmod1"
180 echo=0
190 CALLBIN "brt_pascalinit" (echo) !  Initialization routine.
200 !
210 CALLBIN "pmod1_entpt" (PassParam)
220 !
230 DISP " After binary called. "
240 DISP "PassParam = ";PassParam
250 !
260 CALLBIN "brt_pascalwrap"
270 !
280 SCRATCHBIN "pmod1"
290 END
```

   The `LOADBIN` statement (line 170) links the binary to BASIC. This example assumes that the binary is the file named `bin1` in the current working directory. (If it is not in a file in that directory, then you would need to specify a path name.)

   After executing `LOADBIN` and *before* calling a Pascal binary, you must execute the following statement (line 190):

```
CALLBIN "brt_pascalinit" (echo)
```

   The pass parameter `echo` must be declared to be a BASIC `INTEGER`, and it must have a value of 0.

   The `CALLBIN` statement (line 210) branches to the specified entry point in the binary. In this case, the entry point is named `pmod1_entpt`; the `pmod1` portion matches the module name; the `entpt` portion is the procedure (entry point) name; the underscore character (_) separates the two portions.

After the binary has finished execution, it returns control to the BASIC program. In this example, the BASIC program displays the modified variable **PassParam** is passed *by reference*, which allows the binary to modify the variable's contents. Passing parameters is further described in subsequent sections.

After finishing all calls to a Pascal binary, you should execute a call to the following routine (line 260):

```
CALLBIN "brt_pascalwrap"
```

NOTE

Calls to `brt_pascalwrap` will occasionally fail if a previous call to a Pascal binary failed.

Once you no longer need the binary, you can unlink it from BASIC with the `SCRATCHBIN` statement (line 280). Note that the file is still in the HP-UX file system; however, it is not linked to Technical BASIC any longer, and is therefore inaccessible from BASIC.

Summary of Parameter-Type Matching

Here is a list of all the pass parameter types that you can send to a Pascal binary, along with the corresponding Pascal formal parameter types.

Table 13-1. Matching Parameters in BASIC and Pascal Binaries

BASIC Data Type	Pascal Data Type
Simple `INTEGER`	`integer`
`INTEGER` array	`packed array of integer`
Simple `SHORT`	`real`
`SHORT` array	`packed array of real`
Simple `REAL`	`longreal`
`REAL` array	`packed array of longreal`
Simple string	`packed array of char`
String array	`packed array of char`

Note that arrays must **always** be passed by reference.

Restrictions

You can use almost all the features of a language in a binary program. However, there are a few things that you cannot do with Pascal binaries.

Maximum Number of Binaries

At any one time, there can be only 1 Pascal binary loaded at a time. Note, however, that an individual binary may contain *several* entry points—Pascal `procedures`—as long as they are placed in one Pascal `module`.

NOTE

BASIC **cannot** check to see if there is a Fortran binary currently loaded; therefore, no error will be reported if you try to load a second Fortran binary. Always use `SCRATCHBIN` to unload the current Fortran binary before loading another Fortran binary.

File I/O Restrictions

Binary programs can perform I/O operations on files, with only one restriction: if the binary is to access a file of `text/data` (HP-UX "ASCII"), then the binary must open the file, access the information, and close the file when BASIC is not accessing the file. For instance, if a BASIC program has a particular `text/data` file currently open, then no binary should access that file. After BASIC has closed it, the binary may open it, perform I/O operations on it, and then close it. (The converse situation has the same restriction.)

Note, however, that there is no additional restriction on the number of files that BASIC or Pascal may have open at one time. For instance, BASIC can still have up to 10 files of type `text/data` open simultaneously, while Pascal can have as many as 10 files open at one time.

Examples of Pascal file I/O are given in the last section of this chapter.

HP-UX Environment May Not Be Accessible

HP-UX environment variables, such as `TERM` and `PATH`, may not be accessible to binaries. System calls, such as `ioctl`, should be used with caution.

Standard I/O Streams Are Not Accessible

Binaries should **not** perform operations on the "standard I/O streams"—namely, displaying on the screen (`output` in Pascal) or getting characters from the keyboard (`input` and `keyboard` in Pascal). For example, there is no guarantee that the Pascal standard procedure `writeln` (to `output`) will work in all binaries, although you may get it to work in some instances.

Error Trapping in Binaries

Technical BASIC usually recovers gracefully from errors encountered while using binaries. However, there are some errors that BASIC cannot handle, and thus may even cause the HP-UX system to log you out. Here are some suggestions on how to avoid this type of situation.

1. The best approach to avoiding errors when using binaries is to **thoroughly** test the binary (with a stand-alone program) **before** calling it from BASIC.
2. When you are ready to actually call it from BASIC, make sure that the you match pass parameters correctly: same number of parameters, correct type-matching, and in the right order. (Later sections provide additional examples of passing parameters to Pascal binaries.) Also be sure that the binary stays withing the bounds of data structures and arrays, expecially when the binary uses pointers.
3. If you do get an error while loading or executing a binary, you should fix the problem in the binary first. Then when back in BASIC, you should use `SCRATCHBIN` to unload the old copy before using `LOADBIN` to load the latest copy. This action must be taken, because `LOADBIN` will not load a binary if a binary of the same name is currently loaded.

A Closer Look at Compiling and Linking

The example binary used in this section is the same one used earlier; it doubles the value of an integer that is passed to it.

```
module pmod1;

export
  procedure entpt(var formal_param : integer);

implement
  procedure entpt(var formal_param : integer);
    begin
      formal_param := formal_param * 2;
    end;

end.
```

While in the HP-UX shell, compile the Pascal binary and link it to BASIC. Use the `makebin_p` shell script for this purpose; the default location of this script is in the `/usr/bin` directory. Assuming the Pascal program is called `pmod1.p`, you could type:

```
$  /usr/bin/makebin_p  pmod1 Return
```

Here are the contents of the **makebin_p** shell script (for Series 500 systems):

```
pc  -c  $1.p
ld -rd  -o $1  $1.o  /usr/lib/bprt0.o  -lheap2 -lpc -lb
```

pc	invokes the Pascal compiler.
-c	suppresses the otherwise automatic linking step; that is, the Pascal compiler is told to generate an unlinked, relocatable object file (suffix .o)—the file's name is specified in the next parameter.
$1	specifies that the parameter passed to the script (in this case, pmod1) is to be used here. Thus, the file named pmod1.p is to be compiled. Consequently, the name pmod1.o is given to the relocatable object file.
ld	is the "link editor" command.
-rd	options that indicate (respectively): the specified object file (pmod1.o) is to be loaded as relocatable (re-linkable) code; the definition of "COMMON" storage is to be forced.
-o $1	specifies that the object file is to be named pmod1, rather than given the default name a.out.
$1.o	specifies which file is to be loaded (pmod1.o in this case).
/usr/lib/bprt0.o	is a special version of the crt.o library that is to be searched when linking Pascal binaries to BASIC.
-lheap2	indicates that the /lib/libheap2.a library is to be searched.
-lpc	indicates that the /lib/libpc.a library is to be searched.
-lb	indicates that the /lib/libb.a library is to be searched.

If there are still unresolved references[1] after executing this script, you will have to specify additional libraries:

- if the reference is to a system library (/lib/libxxx.a), then specify it in an ld option, such as -lpc;
- if the reference is to another library, such as the /usr/lib/bprt0.o library above, then you will need to specify it by name.

Now you can enter the Technical BASIC system, and run a BASIC program that loads and calls the binary.

[1] If the **LOADBIN** or **CALLBIN** statements report an error, there are probably unresolved references in the binary. You can use the HP-UX **nm** command to find them.

Example Pascal Binaries

This section contains examples and corresponding explanations of the following tasks:

- Passing parameters between BASIC and Pascal binaries.
    - Simple numerics.
    - Numeric arrays.
    - Simple strings.
    - String arrays.
- Using `text/data` type files for data interchange.

If you have trouble understanding the mechanisms of "passing by reference" or "passing by value", then you may want to study examples of passing parameters in the "Subprograms" section of the "User-Defined Functions and Subprograms" chapter. Additional details about text files are provided in the "Data Storage and Retrieval" chapter section called "Using text/data Files."

Passing Simple Numeric Parameters

There are three BASIC numeric data types: `INTEGER`, `SHORT`, and `REAL`. All of these types can be passed to Pascal binaries. Here is the required correspondence between BASIC pass parameters and Pascal formal parameters:

Table 13-2. Matching Simple Numeric Parameters

BASIC Pass Parameter	Corresponding Pascal Formal Parameter
`INTEGER`	`integer`
`SHORT`	`real`
`REAL`	`longreal`

Here are general rules for passing these types of parameters (examples are given subsequently):

- If a parameter is to be passed **by reference**, then it must be placed in a BASIC variable. The Pascal binary must declare the corresponding parameter as a `var` parameter in the Pascal procedure heading.
- If a parameter is to be passed **by value**, the BASIC parameter must be an expression (such as a numeric literal, a variable enclosed in parentheses, or a combination of numeric items containing numeric operators). The corresponding Pascal formal parameter must not have a `var` declaration.

This BASIC program passes three parameters to the subsequent Pascal binary.

```
100 INTEGER IntVar,echo !  Declare simple numeric types.
110 IntVar=1000000
120 !
130 REAL RealVar !         Redundant, since REAL is default
140 RealVar=8e+307 !        (when type is not declared).
150 !
160 DISP " Before binary called. "
170 DISP "-----------------------------"
180 DISP "Integer variable = ";IntVar
190 DISP "Real variable    = ";RealVar
200 DISP
210 !
220 LOADBIN "pmodns"
230 echo=0
240 CALLBIN "brt_pascalinit" (echo) !  Initialization routine.
250 !
260 CALLBIN "pmodns_ns" (IntVar,1.5,RealVar)
270 !
280 DISP " After binary called. "
290 DISP "-----------------------------"
300 DISP "Integer variable = ";IntVar
310 DISP "Real variable    = ";RealVar
320 !
330 CALLBIN "brt_pascalwrap"
340 !
350 SCRATCHBIN "pmodns"
360 END
```

Here is the Pascal binary.

```
module pmodns;

export

  procedure ns(var intv   : integer;
                   long1  : longreal;
               var long2  : longreal );

implement

  procedure ns(var intv   : integer;
                   long1  : longreal;
               var long2  : longreal);
    begin
      intv  := intv * 2;
      long2 := long1 * long2;
    end;

end.
```

Here are the results of running the program:

```
 Before binary called.
------------------------------
Integer variable =  1000000
Real variable    =  8e+307


 After binary called.
------------------------------
Integer variable =  2000000
Real variable    =  1.2e+308
```

The 1st BASIC pass parameter, `IntVar`, is passed by reference. The corresponding Pascal formal parameter is a `var` parameter of type `integer`. Passing a variable by reference allows the binary to modify that variable's contents, thereby allowing information to be "passed back" to BASIC.

The 2nd BASIC pass parameter, `1.5`, is passed by value since it is an expression. Note that the corresponding Pascal formal parameter has no `var` declaration in the procedure heading.

The 3rd parameter is passed by reference, since it is a variable. The corresponding Pascal formal parameter is a `var` parameter of type `longreal`.

Passing Numeric Array Parameters

There are three BASIC numeric array types: INTEGER, SHORT, and REAL. All of these types can be passed to Pascal binaries. Here is the required correspondence between BASIC pass parameters and Pascal formal parameters:

Table 13-3. Matching Numeric Array Parameters

BASIC Pass Parameter	Corresponding Pascal Formal Parameter
INTEGER array	Array of integer
SHORT array	Array of real
REAL array	Array of longreal

Note that arrays are **always** passed by reference, using BASIC array variables. The Pascal binary must declare the corresponding parameter as a var parameter in the Pascal procedure heading.

The following example passes 2 arrays to a Pascal binary.

```
100  DISP " Before calling binary. "
110  DISP "----------------------------"
120 OPTION BASE 1
130 REAL RealArray(4) ! 4 elements
135 !
140 FOR I=1 TO 4
150   RealArray(I)=5*I
160   DISP "RealArray(";I;") = ";RealArray(I)
170 NEXT I
180 DISP
190 INTEGER IntArray(2,3) !  2 by 3 array.
200 FOR RowN=1 TO 2
210   FOR ColN=1 TO 3
220     IntArray(RowN,ColN)=10*RowN+ColN
230     DISP "IntArray(";RowN;",";ColN;") = ";IntArray(RowN,ColN)
240   NEXT ColN
250 NEXT RowN
260 DISP
270 LOADBIN "pmodan"
280 echo=0
290 CALLBIN "brt_pascalinit" (echo) !  Initialization routine.
300 !
310 CALLBIN "pmodan_arrayn" (RealArray(),IntArray()) ! Always by reference.
320 !
330 DISP " After calling binary. "
340 DISP "----------------------------"
350 FOR I=1 TO 4
360   DISP "RealArray(";I;") = ";RealArray(I)
370 NEXT I
380 DISP
390 FOR RowN=1 TO 2
400   FOR ColN=1 TO 3
410     DISP "IntArray(";RowN;",";ColN;") = ";IntArray(RowN,ColN)
420   NEXT ColN
430 NEXT RowN
440 !
450 CALLBIN "brt_pascalwrap"
460 SCRATCHBIN "pmodan"
470 END
```

Here is an example Pascal binary that would work with the preceding BASIC CALLBIN statement.

```
module pmodan;

export

  type
    lrealarr_type = packed array [1..4] of longreal;
    intarr_type   = packed array [1..2, 1..3] of integer;

  procedure arrayn(var lrealarr : lrealarr_type;
                   var intarr   : intarr_type);

implement

  procedure arrayn(var lrealarr : lrealarr_type;
                   var intarr   : intarr_type);
    var
      element,
      row, col  : integer;

    begin
      for element := 1 to 4 do
        lrealarr[element] := lrealarr[element] + 100;

      for row := 1 to 2 do
        for col := 1 to 3 do
          intarr[row,col] := intarr[row,col] + 100*row;
    end;

end.
```

Here are the results of running the BASIC program:

```
 Before calling binary.
----------------------------
RealArray( 1 ) =  5
RealArray( 2 ) =  10
RealArray( 3 ) =  15
RealArray( 4 ) =  20

IntArray( 1 , 1 ) =  11
IntArray( 1 , 2 ) =  12
IntArray( 1 , 3 ) =  13
IntArray( 2 , 1 ) =  21
IntArray( 2 , 2 ) =  22
IntArray( 2 , 3 ) =  23

 After calling binary.
----------------------------
RealArray( 1 ) =  105
RealArray( 2 ) =  110
RealArray( 3 ) =  115
RealArray( 4 ) =  120

IntArray( 1 , 1 ) =  111
IntArray( 1 , 2 ) =  112
IntArray( 1 , 3 ) =  113
IntArray( 2 , 1 ) =  221
IntArray( 2 , 2 ) =  222
IntArray( 2 , 3 ) =  223
```

The binary assumes that the calling BASIC program will send a real array with 4 elements and a 2×3 integer array. A more general method would be to pass arrays of variable sizes to the binary. In such cases, the calling program could communicate the size of the array; here are two possible methods:

- By passing parameter(s) that indicate the number of elements (and dimensions).
- By assigning a unique "flag" value to an array element to indicate that it is the last element in the array.

Passing Simple String Parameters

Here is the required correspondence between simple BASIC string pass parameters and Pascal formal parameters:

Table 13-4. Matching Simple String Parameters

BASIC Pass Parameter	Corresponding Pascal Formal Parameter
Simple string	Packed array of `char`

Here are general rules for passing these types of parameters (examples are given subsequently):

- If a parameter is to be passed **by reference**, then it must be placed in a BASIC variable. The Pascal binary must declare the corresponding parameter as a `var` parameter in the Pascal procedure heading. Note that even though a Pascal binary may change a BASIC string variable's contents, it cannot change the BASIC string variable's length. (This topic is discussed in the section called "BASIC String Length Headers".)
- If a parameter is to be passed **by value**, the BASIC parameter must be an expression (such as a quoted string literal, a variable enclosed in parentheses, or a combination of string items containing the string concatenation operator `&`). The corresponding Pascal formal parameter must not have a `var` declaration.

The following BASIC program and Pascal binary illustrate passing simple string parameters.

```
100 DISP " Before binary called."
110 DISP "---------------------------"
120 DIM StrVar$[18] !   Simple string of 18 chars (default length).
130 StrVar$="BASIC string value"
140 DISP "StrVar$ = ";StrVar$
150 DISP
160 LOADBIN "pmodss"
170 echo=0
180 CALLBIN "brt_pascalinit" (echo) !  Initialization routine.
190 !
200 CALLBIN "pmodss_simples" (StrVar$,StrVar$&"")
210 DISP " After binary called."
220 DISP "---------------------------"
230 DISP "StrVar$ = ";StrVar$
240 DISP
250 CALLBIN "brt_pascalwrap"
260 SCRATCHBIN "pmodss"
270 END
```

Here is the corresponding Pascal binary.

```
module pmodss;

export

  type
    simple_str_type = packed array [1..18] of char;

  procedure simples(var simple_str1 : simple_str_type;
                        simple_str2 : simple_str_type);

implement

  procedure simples(var simple_str1 : simple_str_type;
                        simple_str2 : simple_str_type);
    begin
      simple_str1 := 'new characters....';
      simple_str2 := 'new characters too';
    end;

end.
```

Here are the results of running the program.

```
 Before binary called.
----------------------------
StrVar$ = BASIC string value

 After binary called.
----------------------------
StrVar$ = new characters....
```

The 1st parameter was passed by reference, and the 2nd was passed by value (the concatenation operation formed a string expression, which is always passed by value). That is why the Pascal assignment to the first variable, `simple_str1`, changed the value of the BASIC variable called `StrVar$`; but the assignment to `simple_str2` didn't.

BASIC String Length Headers

BASIC string *variables* have a *length header* that indicates how many characters the variable *currently* contains. However, this header is not accessible to Pascal binaries. (Pascal string variables have their own header, but it is not set to match the BASIC header.) Thus, Pascal binaries **cannot** modify the BASIC string variable's length.

For instance, suppose that you pass a string variable (by reference) to a Pascal binary. The binary then proceeds to change the length of the string; however, it *cannot* modify the BASIC string variable's length header. Thus upon returning to BASIC, there is no indication that the length of the string variable is any different than when it was passed to the binary.

There are two steps in the general work-around for this type of situation:

1. Before passing the variable (by reference), pad the string with blank characters to the *maximum* length of string that the binary can return. For instance, the following statement pads the BASIC string variable with trailing blanks and sets its length to the maximum (dimensioned) length.

   ```
   ByRef$[LEN(ByRef$)+1]=" "
   ```

   Note that this particular statement will cause an error if the string length is already equal to the maximum (dimensioned) length.

2. After returning to the BASIC program, determine the string variable's new length.

    a. If a string length parameter is passed (by reference) to the binary, then the binary can modify the string and corresponding length parameter accordingly and then pass it back to BASIC. Here is an example of this technique.

```
100 String$="123456"
110 Length=LEN(String$)
120 CALLBIN strmod(String$,Length)

procedure strmod(var strvar    : string_type;
                 var strlength : integer);
  begin
    strlength := strlen(strvar) / 2;   (* Cut length in half.    *)
    strvar := str(strvar,1,strlength); (* Use chars 1..strlength.*)
  end;

130 String$=String$[1,Length]
```

    b. If the binary has a special character to mark the end of the string, such as CHR$(0), then BASIC can search the string and set the string length to 1 less than the position of the null. Here is an example of this technique.

```
NullPos=POS(ByRef$,CHR$(0))
ByRef$=ByRef$[1,NullPos-1]
```

Passing String Arrays

Here is the required correspondence between BASIC string-array pass parameters and Pascal formal parameters:

Table 13-5. Matching String Array Parameters

BASIC Pass Parameter	Corresponding Pascal Formal Parameter
String array	Packed array of `char`

Note that arrays are **always** passed by reference, using BASIC array variables. The Pascal binary must declare the corresponding parameter as a `var` parameter in the Pascal procedure heading.

Note that even though a Pascal binary may change a BASIC string variable's contents, it cannot change the BASIC string variable's length. (This topic is discussed in the preceding section called "BASIC String Length Headers".)

Here is an example of passing a string array to a Pascal binary.

```
100 DIM StrArr$(4)[5] !  5 elements, 5 chars each.
110 StrArr$(0)="What "
120 StrArr$(1)="is   "
130 StrArr$(2)="your "
140 StrArr$(3)="name "
150 StrArr$(4)="?"
160 DISP " Before calling binary."
170 DISP "--------------------------"
180 FOR I=0 TO 4
190   DISP StrArr$(I);"/";
200 NEXT I
210 DISP @ DISP
220 !
230 LOADBIN "pmodas"
240 echo=0
250 CALLBIN "brt_pascalinit" (echo) !  Initialize.
260 !
270 CALLBIN "pmodas_arrays" (StrArr$())
280 !
290 DISP " After calling binary."
300 DISP "--------------------------"
310 FOR I=0 TO 4
320   DISP StrArr$(I);"/";
330 NEXT I
340 DISP @ DISP
350 !
360 CALLBIN "brt_pascalwrap"
370 SCRATCHIN "pmodas"
380 END
```

Here is an example of a Pascal binary that would work with the preceding CALLBIN statement.

```
module pmodas;

export

  type
    str_array_type = packed array [1..5,1..5] of char;

  procedure arrays(var pass_array : str_array_type);

implement

  procedure arrays(var pass_array : str_array_type);
    var temp_array : str_array_type;
        row,col    : integer;

    begin
      (* Make backup copy of words to be changed. *)
      for row:=1 to 4 do
        temp_array[row] := pass_array[row];

      (* Now rearrange words. *)
      pass_array[1] := temp_array[3];
      pass_array[2] := temp_array[4];
      pass_array[3] := temp_array[2];
      pass_array[4] := temp_array[1];
    end;

end.
```

Here are the results of running the program.

```
 Before calling binary.
-------------------------
What /is /your /name /?/

 After calling binary.
-------------------------
your /name /is  /What/?/
```

Note once again that the Pascal binary modifies some of the string variables' lengths; however, the BASIC program variables' string lengths are **not** modified. (See the preceding section for a work-around.)

Using Files with Pascal Binaries

BASIC programs and Pascal binaries can also communicate via text files. The main restriction is that BASIC and Pascal use the file *independently*. That is, if BASIC opens and accesses a file, then it should close the file before the binary attempts to open the file. The converse situation has the same restrictions. Note, however, that there is no additional restriction on the number of files that BASIC or the Pascal binary may have open at any one time. For instance, BASIC can still have up to 10 files of type `text/data` open simultaneously.

Here is an example of a BASIC program opening a text file, putting some words into it, then closing it before calling a binary. The binary then opens the same file, modifies its contents, and returns control to BASIC.

```
100 DIM StrArr$(4)[5] !      String array (5 elements, 5 chars each).
110 StrArr$(0)="What " !     Pad strings to max. length.
120 StrArr$(1)="is   "
130 StrArr$(2)="your "
140 StrArr$(3)="name "
150 StrArr$(4)="?    "
160 !
170 ASSIGN 11 TO "text_file" !  Open text file (BASIC will
180 FOR I=0 TO 4 !               create it, if non-existent).
190   OUTPUT 11 ; StrArr$(I) !  Write string items into file.
200 NEXT I
210 CLEAR !                     Clear screen.
220 DISP " Before calling binary."
230 DISP "-------------------------"
240 ASSIGN 11 TO "text_file" ! Reset file pointer (for read).
250 FOR I=0 TO 4
260   ENTER 11 ; StrArr$(I) !  Read string items from file.
270   DISP StrArr$(I);"/";
280 NEXT I
290 DISP @ DISP
300 ASSIGN 11 TO "*" !         Close file.
310 !
320 LOADBIN "pmodf"
330 echo=0
340 CALLBIN "brt_pascalinit" (echo) !  Initialization routine.
350 !
```

```
360 FileName$="text_file"
370 CALLBIN "pmodf_tf" (FileName$)
380 !
390 DISP " After calling binary."
400 DISP "------------------------"
410 ASSIGN 11 TO "text_file" ! Reset file pointer.
420 FOR I=0 TO 4
430   ENTER 11 ; StrArr$(I) !  Read string items from file.
440   DISP StrArr$(I);"/";
450 NEXT I
460 ASSIGN 11 TO "*" !         Close file.
470 DISP @ DISP
480 !
490 CALLBIN "brt_pascalwrap"
500 SCRATCHBIN "pmodf"
510 END
```

Here is an example Pascal binary that would work with the preceding BASIC program.

```
module pmodf;

export

  type
    pathname_type = packed array [1..18] of char;

  procedure tf(var file_name : pathname_type);

implement

  procedure tf(var file_name : pathname_type);
    var file_var   : text;
        from_file,
        to_file    : packed array [1..5,1..5] of char;
        row        : integer;

    begin
      (* Open file for reading. *)
      reset(file_var,file_name);

      (* Read lines of file. *)
      for row:=1 to 5 do
        readln(file_var,from_file[row]);
        close(file_var);

        (* Now rearrange words. *)
        to_file[1] := from_file[3];
        to_file[2] := from_file[4];
        to_file[3] := from_file[2];
        to_file[4] := from_file[1];
        to_file[5] := from_file[5];

        (* Then rewrite file. *)
        rewrite(file_var,file_name);
        for row:=1 to 5 do
          writeln(file_var,to_file[row]);

        close(file_var);

      end;

  end.
```

Here are the results of the BASIC program calling the Pascal binary.

```
   Before calling binary.
  -------------------------
  What /is   /your /name /?    /


   After calling binary.
  -------------------------
  your /name /is   /What /?    /
```

FORTRAN Binaries 14

Introduction

Generally, you will be using the Technical BASIC system to execute programs written in the Technical BASIC language. However, you can also write programs in the FORTRAN language and then call (execute) them from Technical BASIC. In this manual, such programs are termed binary programs. The term "binary" was probably coined because the programs written in another language and compiled into executable object code cannot be easily read by humans from the BASIC system—they look like just a bunch of binary patterns.

Binary programs are useful in the following situations:

- An application is already written in FORTRAN, and you don't want or have time to translate it into Technical BASIC code.
- FORTRAN supports a feature that is not available in Technical BASIC, or the FORTRAN version runs faster that the BASIC version.

Chapter Contents

Tasks/Topics	Page
Overview	14-2
Structure of FORTRAN programs and binaries	14-2
Compiling and linking FORTRAN binaries	14-3
Summary of parameter type matching	14-5
Restrictions	14-6
A closer look at compiling and linking	14-8
Examples of passing parameters	14-10
Examples of file I/O	14-22

Overview

This section briefly covers the following topics. It is intended to quickly give you a global view of using FORTRAN binaries. Specific details of each topic are presented in the subsequent sections.

- Structure of FORTRAN programs and binaries.
- Procedure for creating FORTRAN binaries.
- Parameter-type matching.
- Restrictions on FORTRAN binaries.

Structure of FORTRAN Programs and Binaries

Here is a FORTRAN program that contains (and uses) a SUBROUTINE. The unshaded portion (the subroutine) can be used up as a binary—that is, it can be separately compiled and linked to BASIC, and then called as a separate entry point by a BASIC program.

```
PROGRAM callbin1

INTEGER*4 pass_param

pass_param=7
PRINT *,"Before calling 'entry_pt':"
PRINT *,"pass_param=",pass_param

CALL entry_pt(pass_param)

PRINT *,"After  calling 'entry_pt':"
PRINT *,"pass_param=",pass_param

END


SUBROUTINE entry_pt(formal_param)

INTEGER*4 formal_param

formal_param = formal_param * 2

RETURN
END
```

Here are the results of running the program:

```
Before calling 'entry_pt':
pass_param=7
After  calling 'entry_pt':
pass_param=14
```

Note that the FORTRAN subroutine named **entry_pt** was able to modify the variable **pass_param**'s contents, since the parameter was passed by reference. (Note that **all** parameters must be passed by reference to FORTRAN binaries, since FORTRAN does not support pass by value.)

Compiling and Linking

1. If the FORTRAN code is structured as a program, you will need to re-structure it as subroutine(s)—one for each entry point. Store this code in a file separate from the original copy. In the above example, this is the SUBROUTINE named entry_pt.

```
SUBROUTINE entry_pt(formal_param)

INTEGER*4 formal_param

formal_param = formal_param * 2

RETURN
END
```

NOTE

You should have, at this point, thoroughly debugged the FORTRAN subroutine by calling it from a FORTRAN program (as in the example FORTRAN program above). The reason for this approach is that even though BASIC attempts to trap errors, it cannot trap them all.

2. While in the HP-UX shell, compile the FORTRAN binary and link it to any libraries that are required to resolve the external references. A shell script is provided for this purpose. When you use the script, specify the name of the file in which you want the binary to be stored. For instance, if the binary is in a file named bin1.f, then the script call would be:

```
$ /usr/bin/makebin_f  bin1 Return
```

3. Enter the Technical BASIC system, and type in and run a BASIC program that loads and calls the binary. Make sure that the parameters passed to the binary match those expected by the binary—in both number and in type of each parameter. (See the subsequent table for a complete list of correspondence between BASIC and FORTRAN parameters.)

   Here is an example BASIC program that calls the preceding binary:

```
100 INTEGER PassParam !  Declare simple numeric type.
110 PassParam=7
120 !
130 DISP " Before binary called. "
140 DISP "PassParam = ";PassParam
150 DISP
160 !
170 LOADBIN "bin1"
180 !
190 CALLBIN "f_init" !  Initialization routine.
200 !
210 CALLBIN "entry_pt" (PassParam)
220 !
230 DISP " After binary called. "
240 DISP "PassParam = ";PassParam
250 !
260 SCRATCHBIN "bin1"
270 END
```

   The first part of the program (lines 100 to 150) declare an `INTEGER` type variable, assign it a value of 7, and then display its contents.

   The `LOADBIN` statement (line 170) links the binary to BASIC. This example assumes that the binary is the file named `bin1` in the current working directory. (If it is not in a file in that directory, then you would need to specify a path name.)

   After executing `LOADBIN` and *before* calling a FORTRAN binary, you must execute the following statement (line 190):

```
CALLBIN "f_init"
```

   The `CALLBIN` statement (line 210) branches to the specified entry point in the binary. In this case, the entry point is named `entry_pt`.

   After the binary has finished execution, it returns control to the BASIC program. In this example, the BASIC program displays the modified value of the variable `PassParam`. Note that the BASIC binary to modify the variable's contents. Passing parameters is further described in subsequent sections.

Once you no longer need the binary, you can unlink it from BASIC with the SCRATCHBIN statement (line 260). Note that the file is still in the HP-UX file system; however, it is not linked to Technical BASIC, and is therefore inaccessible from BASIC.

Summary of Parameter-Type Matching

Here is a list of all the pass parameter types that you can send to a FORTRAN binary, along with the corresponding FORTRAN formal parameter types.

Table 14-1. Matching BASIC Pass Parameters with FORTRAN Formal Parameters

BASIC Data Type	FORTRAN Data Type
Simple INTEGER	INTEGER*4
INTEGER array	Array of INTEGER*4
Simple SHORT	REAL
SHORT array	Array of REAL
Simple REAL	DOUBLE PRECISION
REAL array	Array of DOUBLE PRECISION
Simple string	Series 200/300: CHARACTER **string_length*
	Series 500: INTEGER vector (1-dimensional array)
String array	Series 200/300: Array of CHARACTER **string_length*
	Series 500: INTEGER array

Note that **all** parameters are passed by reference with FORTRAN, since the language does not support passing parameters by value.

Restrictions

You can use almost all the features of a language in a binary program. However, there are a few things that you cannot do with FORTRAN binaries.

Maximum Number of Binaries

At any one time, there can be only 1 FORTRAN binary loaded at a time. Note, however, that an individual binary may contain *several* entry points—FORTRAN subroutines—as long as they are placed in one object file.

NOTE

BASIC **cannot** check to see if there is a FORTRAN binary currently loaded; therefore, no error will be reported if you try to load a second FORTRAN binary. Always use `SCRATCHBIN` to unload the current FORTRAN binary before loading another FORTRAN binary.

File I/O Restrictions

Binary programs can perform I/O operations on files, with only one restriction: if the binary is to access a file of `text/data` (HP-UX "ASCII"), then the binary must open the file, access the information, and close the file while BASIC is not accessing the file. For instance, if a BASIC program has a particular `text/data` file currently open, then no binary should access that file. After BASIC has closed it, the binary may open it, perform I/O operations on it, and then close it. (The converse situation has the same restriction.)

Note, however, that there is no additional restriction on the number of files that BASIC or FORTRAN may have open at one time. For instance, BASIC can still have up to 10 files of type `text/data` open simultaneously. FORTRAN binaries can have as many files open as allowed by the HP-UX system.

Examples of FORTRAN file I/O are given in the last section of this chapter.

HP-UX Environment May Not Be Accessible

HP-UX environment variables, such as `TERM` and `PATH`, may not be accessible. System calls, such as `ioctl`, should be used with caution.

Standard I/O Streams Are Not Accessible

Binaries should **not** perform operations on the "standard I/O streams"—namely, displaying on the screen (unit 6 in FORTRAN) or getting characters from the keyboard (unit 5 in FORTRAN). For example, there is no guarantee that the FORTRAN standard subroutine `PRINT` (to unit 6) will work in all binaries, although you may get it to work in some instances.

Error Trapping in Binaries

Technical BASIC usually recovers gracefully from errors encountered while using binaries. However, there are some errors that BASIC cannot handle, and thus may even cause the HP-UX system to log you out. Here are some suggestions on how to avoid this type of situation.

1. The best approach to avoiding errors when using binaries is to **thoroughly** test the binary (with a stand-alone program) **before** calling it from BASIC.
2. When you are ready to actually call it from BASIC, make sure that the you match pass parameters correctly: same number of parameters, correct type-matching, and in the right order. (Later sections provide additional examples of passing parameters to FORTRAN binaries.) You also need to be careful to stay within the bounds of arrays while in the binary. Also note that BASIC uses *row-major* order in arrays, while FORTRAN uses *column-major* order. This results in the array subscripts being reversed when going from BASIC to FORTRAN, and vice versa. (See the discussions of passing arrays in the "Example FORTRAN Binaries" section.)
3. If you do get an error while loading or executing a binary, you should fix the problem in the binary first. Then when back in BASIC, you should use `SCRATCHBIN` to unload the old copy before using `LOADBIN` to load the latest copy. This action must be taken, because `LOADBIN` will not load a binary if a binary of the same name is currently loaded.

A Closer Look at Compiling and Linking

The example FORTRAN binary used in this section is the same one used earlier; it doubles the value of an integer that is passed to it.

```
SUBROUTINE entry_pt(formal_param)

INTEGER*4 formal_param

formal_param = formal_param * 2

RETURN
END
```

While in the HP-UX shell, compile the FORTRAN binary and link it to BASIC. Use the **makebin_f** shell script for this purpose; the default location of this script is in the **/usr/bin** directory. Assuming the FORTRAN program is called **bin1.f**, you could type:

```
$  /usr/bin/makebin_f  bin1  [Return]
```

Here are the contents of the `makebin_f` shell script (for Series 500 systems):

```
for filef
do
  fc  -c     $filef.f
  ld  -rd  -o $filef  $filef.o  /usr/lib/bfrt0.o  -lI77 -lF77 -lm -lc
done
```

`fc`	invokes the FORTRAN compiler.
`-c`	suppresses the otherwise automatic linking step; that is, the FORTRAN compiler is told to generate an unlinked, relocatable object file (suffix `.o`)—the file's name is specified in the next parameter.
`$filef.f`	specifies that the parameter passed to the script (in this case, `bin1`) is to be used here. Thus, the file named `bin1.f` is to be compiled. Consequently, the name `bin1.o` is given to the relocatable object file.
`ld`	is the "link editor" command.
`-rd`	options that indicate (respectively): the specified object file (`bin1.o`) is to be loaded as relocatable (re-linkable) code; the definition of "COMMON" storage is to be forced.
`-o $filef.o`	specifies which file is to be loaded (`bin1.o` in this case).
`/usr/lib/bfrt0.o`	a special version of the `crt.o` library that must be searched when linking FORTRAN binaries to BASIC.
`-lI77`	indicates that the `/lib/libI77.a` library is to be searched.
`-lF77`	indicates that the `/lib/libF77.a` library is to be searched.
`-lm`	indicates that the `/lib/libm.a` library is to be searched.
`-lc`	indicates that the `/lib/libc.a` library is to be searched.

If there are still unresolved references[1] after executing this script, you will have to specify additional libraries:

- if the reference is to a system library (`/lib/libxxx.a`), then specify it in an `ld` option, such as `-lm`;
- if the reference is to another library, such as the `/usr/lib/bfrt0.o` library above, then you will need to specify it by name.

[1] If the **LOADBIN** or **CALLBIN** statements report an error, there are probably unresolved references in the binary. You can use the HP-UX **nm** command to find them.

Examples of FORTRAN Binaries

This section contains examples and explanations of the following tasks:

- Passing parameters between BASIC and FORTRAN binaries.
  - Simple numerics
  - Numeric arrays
  - Simple strings
  - String arrays
- Using `text/data` type files for data interchange between BASIC and FORTRAN binaries.

If you have trouble understanding the mechanism of "passing by reference", then you may want to study further examples of passing parameters in the "Subprograms" section of the "User-Defined Functions and Subprograms" chapter. Additional details of using `text/data` files are provided in the "Data Storage and Retrieval" chapter section called "Using text/data Files".

Passing Simple Numeric Parameters

There are three BASIC numeric data types: `INTEGER`, `SHORT`, and `REAL`. You can pass all of these types to FORTRAN binaries. Here is the required correspondence between BASIC pass parameters and FORTRAN formal parameters:

Table 14-2. Matching Simple Numeric Parameters

BASIC Pass Parameter	Corresponding FORTRAN Formal Parameter
`INTEGER`	`INTEGER *4`
`SHORT`	`REAL`
`REAL`	`DOUBLE PRECISION`

All BASIC parameters must be passed **by reference** to FORTRAN binaries. (FORTRAN binaries do not support pass by value.)

This BASIC program passes three parameters to the subsequent FORTRAN binary.

```
100 INTEGER IntVar !  Declare simple numeric types.
110 IntVar=1000000
120 !
130 REAL RealVar !         Redundant, since REAL is default
140 RealVar=8e+307 !        (when type is not declared).
150 !
160 DISP " Before binary called. "
170 DISP "-----------------------------"
180 DISP "Integer variable = ";IntVar
190 DISP "Real variable    = ";RealVar
200 DISP
210 !
220 LOADBIN "fsn"
230 CALLBIN "f_init" !  Initialization routine.
240 !
250 CALLBIN "fsimplen" (IntVar,RealVar)
260 !
270 DISP " After binary called. "
280 DISP "-----------------------------"
290 DISP "Integer variable = ";IntVar
300 DISP "Real variable    = ";RealVar
310 !
320 SCRATCHBIN "fsn"
330 END
```

Here is the FORTRAN binary program.

```
      SUBROUTINE fsimplen(Int,Real)

      INTEGER*4 Int
      DOUBLE PRECISION Real

      Int=2*Int
      Real=2*Real

      RETURN
      END
```

Here are the results of running the program:

```
 Before binary called.
------------------------------
Integer variable =  1000000
Real variable    =  8e+307

 After binary called.
------------------------------
Integer variable =  2000000
Real variable    =  1.6e+308
```

Both BASIC pass parameters, IntVar and RealVar, are passed by reference. The corresponding FORTRAN formal parameters are of type INTEGER*4 and DOUBLE PRECISION, respectively. Passing a variable by reference allows the binary to modify that variable's contents, thereby allowing information to be "passed back" to BASIC.

Passing Numeric Array Parameters

There are three BASIC numeric data types: INTEGER, SHORT, and REAL. You can pass all of these types to FORTRAN binaries. Here is the required correspondence between BASIC pass parameters and FORTRAN formal parameters:

Table 14-3. Matching Numeric Array Parameters

BASIC Pass Parameter	Corresponding FORTRAN Formal Parameter
INTEGER array	INTEGER *4 array
SHORT array	REAL array
REAL array	DOUBLE PRECISION array

All BASIC parameters must be passed **by reference** to FORTRAN binaries. (FORTRAN binaries do not support pass by value.)

Here is an example that passes 2 arrays to a FORTRAN binary.

```
100 OPTION BASE 1
110 DISP " Before calling binary. "
120 DISP "----------------------------"
130 REAL RealArray(4)
140 FOR I=1 TO 4
150   RealArray(I)=I
160   DISP "RealArray(";I;") = ";RealArray(I)
170 NEXT I
180 DISP
190 INTEGER IntArray(2,3)
200 FOR BASICrow=1 TO 2
210   FOR BASICcol=1 TO 3
220     IntArray(BASICrow,BASICcol)=10*BASICrow+BASICcol
230     DISP "IntArray(";BASICrow;",";BASICcol;") = ";
240     DISP IntArray(BASICrow,BASICcol)
```

```
250   NEXT BASICcol
260 NEXT BASICrow
270 DISP
280 LOADBIN "fan"
290 CALLBIN "f_init" !  Initialization routine.
300 !
310 CALLBIN "farrayn" (RealArray(),IntArray()) ! Always pass by ref.
320 !
330 DISP " After calling binary. "
340 DISP "----------------------------"
350 FOR I=1 TO 4
360   DISP "RealArray(";I;") = ";RealArray(I)
370 NEXT I
380 DISP
390 FOR BASICrow=1 TO 2
400   FOR BASICcol=1 TO 3
410     DISP "IntArray(";BASICrow;",";BASICcol;") = ";
420     DISP IntArray(BASICrow,BASICcol)
430   NEXT BASICcol
440 NEXT BASICrow
450 !
460 SCRATCHBIN "fan"
470 END
```

Here is an example FORTRAN binary program that would work with the preceding BASIC CALLBIN statement.

```
      SUBROUTINE farrayn(RealArray,IntArray)

      DOUBLE PRECISION RealArray(4)
      INTEGER*4 IntArray(3,2) !   Note reversal of subscripts!!
      INTEGER BASICrow,BASICcol

      DO 20 BASICrow=1,2
        DO 10 BASICcol=1,3
        IntArray(BASICcol,BASICrow)=IntArray(BASICcol,BASICrow)
     :   +100*BASICrow
 10   CONTINUE
 20   CONTINUE

      RETURN
      END
```

Here are the results of running the BASIC program:

```
  Before calling binary.
-----------------------------
RealArray( 1 ) =  1
RealArray( 2 ) =  2
RealArray( 3 ) =  3
RealArray( 4 ) =  4

IntArray( 1 , 1 ) =  11
IntArray( 1 , 2 ) =  12
IntArray( 1 , 3 ) =  13
IntArray( 2 , 1 ) =  21
IntArray( 2 , 2 ) =  22
IntArray( 2 , 3 ) =  23

  After calling binary.
-----------------------------
RealArray( 1 ) =  1
RealArray( 2 ) =  2
RealArray( 3 ) =  3
RealArray( 4 ) =  4

IntArray( 1 , 1 ) =  111
IntArray( 1 , 2 ) =  112
IntArray( 1 , 3 ) =  113
IntArray( 2 , 1 ) =  221
IntArray( 2 , 2 ) =  222
IntArray( 2 , 3 ) =  223
```

This example binary assumes that the calling BASIC program will send a `REAL` array with 4 elements and a 2×3 `INTEGER` array. A more general method would be to pass arrays of variable sizes to the binary. In such cases, the calling program can communicate the size of the array to the binary. Here are two possible methods:

- By passing parameter(s) that indicate the number of elements (and dimensions).
- By assigning a unique "flag" value to an array element to indicate that it is the last element in the array.

Passing Simple String Parameters

Here is the required correspondence between simple BASIC string pass parameters and FORTRAN formal parameters:

Table 14-4. Matching Simple String Parameters

BASIC Pass Parameter	Corresponding FORTRAN Formal Parameter
Simple string	Series 200/300: `CHARACTER` **string length* Series 500: `INTEGER` vector[1]

All BASIC parameters must be passed **by reference** to FORTRAN binaries. (FORTRAN binaries do not support pass by value.)

A FORTRAN binary may change a string variable's contents but it cannot change the variable's string-length header. This topic is discussed in the section called "BASIC String-Length Headers."

1 A vector is a 1-dimensional array.

Series 200/300 Example

The following BASIC program and Series 200/300 FORTRAN binary illustrate passing simple string parameters.

```
100 DISP " Before binary called."
110 DISP "----------------------------"
120 DIM StrVar$[20] !  Simple string of 20 chars.
130 StrVar$="BASIC   string value"
140 DISP "StrVar$ = ";StrVar$
150 DISP "Length =  ";LEN(StrVar$)
160 DISP
170 LOADBIN "fss3"
180 CALLBIN "f_init" !  Initialization routine.
190 CALLBIN "str300" (StrVar$)
200 DISP " After binary called."
210 DISP "----------------------------"
220 DISP "StrVar$ = ";StrVar$
230 DISP "Length =  ";LEN(StrVar$)
240 DISP
250 SCRATCHBIN "fss3"
260 END
```

Here is a Series 200/300 FORTRAN binary that works with the preceding BASIC program.

```
      SUBROUTINE str300(StringVar)
C
      CHARACTER *20 StringVar
C
      StringVar = 'FORTRAN' // StringVar(8:20)

      RETURN
      END
```

Here are the results of running the program.

```
 Before binary called.
------------------------------
StrVar$ = BASIC   string value
Length =   20

 After binary called.
------------------------------
StrVar$ = FORTRAN string value
Length =   20
```

Series 500 Example

The following BASIC program and Series 500 FORTRAN binary illustrate passing simple string parameters.

```
100 DISP " Before binary called."
110 DISP "----------------------------"
120 DIM StrVar$[14]
130 StrVar$="BASIC   string"
140 StrLen=LEN(StrVar$)
150 DISP "StrVar$ = ";StrVar$
160 DISP "Length =  ";StrLen
170 DISP
180 LOADBIN "fss5"
190 CALLBIN "f_init" !  Initialization routine.
200 CALLBIN "str500" (StrVar$,StrLen)
210 !
220 DISP " After binary called."
230 DISP "----------------------------"
240 DISP "StrVar$ = ";StrVar$
250 DISP "Length =  ";LEN(StrVar$)
260 DISP
270 SCRATCHBIN "fss5"
280 END
```

Here is a Series 500 FORTRAN binary that works with the preceding BASIC program.

```
      SUBROUTINE str500(IntVector,StringLen)

C     IntVector can hold 16 characters:
C     (4 characters per 32-bit integer element).
      INTEGER IntVector(4),StringLen,CopyVector(4)

      CHARACTER *16 StringVar
      EQUIVALENCE (CopyVector,StringVar)
```

```
C     Set length of string variable (16 spaces).
      StringVar = '                '

C     Now copy IntVector into StringVar.
      DO 20 I=1,4
        CopyVector(I) = IntVector(I)
 20   CONTINUE

C     Now perform string manipulation.
      StringVar = 'FORTRAN' // StringVar(8:14)

C     Now copy string back into pass parameter.
      DO 30 I=1,4
        IntVector(I)=CopyVector(I)
 30   CONTINUE

      RETURN
      END
```

Here are the results of running the program.

```
 Before binary called.
-------------------------
StrVar$ = BASIC   string
Length =   14

 After binary called.
-------------------------
StrVar$ = FORTRAN string
Length =   14
```

BASIC String-Length Headers

BASIC strings have a *length header* that indicates how many characters the string *currently* contains; however, this header is not passed to a binary when the string is passed. Thus, FORTRAN binaries **cannot** modify the BASIC string variable's length.

For instance, suppose that you pass a string variable (by reference) to a FORTRAN binary. The binary then proceeds to change the length of the string, but it *cannot* modify the BASIC string's length header. Upon returning to BASIC, there is no indication that the length of the string variable is any different than when it was passed to the binary.

There are two steps in the general work-around for this type of situation:

1. Before passing the variable (by reference), pad the string with blank characters to the *maximum* length of string that the binary can return. For instance, the following statement pads the BASIC string variable with trailing blanks and sets its length to the maximum (dimensioned) length.

   ```
   ByRef$[LEN(ByRef$)+1]=" "
   ```

   Note that this particular statement will cause an error if the string length is already equal to the maximum (dimensioned) length.

2. After returning to the BASIC program, determine the string's new length.

   a. If a string length parameter is passed (by reference) to the binary, then the binary can modify the string and corresponding length parameter accordingly and then pass it back to BASIC. Here is an example of this technique.

   ```
   100 String$="123456"
   110 Length=LEN(String$)
   120 CALLBIN strmod(String$,Length)


        SUBROUTINE strmod(strvar, strlength);

        CHARACTER *18 strvar
        INTEGER strlength

   C    Cut length in half.
        strlength := strlength / 2
   C    Use chars 1..strlength
        strvar =  strvar(1:strlength)
   C
        RETURN
        END


   130 String$=String$[1,Length]
   ```

   b. If the binary has a special character to mark the end of the string, such as `CHR$(0)`, then BASIC can search the string and set the string length to 1 less than the position of the null. Here is an example of this technique.

   ```
   NullPos=POS(ByRef$,CHR$(0))
   ByRef$=ByRef$[1,NullPos-1]
   ```

Passing String Arrays

Here is the required correspondence between BASIC string-array pass parameters and FORTRAN formal parameters:

Table 14-5. Matching String Array Parameters

BASIC Pass Parameter	Corresponding FORTRAN Formal Parameter
String array	Series 200/300: Array of `CHARACTER` **string length* Series 500: `INTEGER` array[1] (with 2 dimensions)

All BASIC parameters must be passed **by reference** to FORTRAN binaries. (FORTRAN binaries do not support pass by value.)

A FORTRAN binary may change a string variable's contents, but it cannot change the variable's string-length header. This topic is discussed in the preceding section called "BASIC String-Length Headers."

Series 200/300 Example

Here is an example of passing a string array to a Series 200/300 FORTRAN binary.

```
100 DIM StrArr$(4)[5] !  5 elements, 5 chars each.
110 StrArr$(0)="What "
120 StrArr$(1)="is   "
130 StrArr$(2)="your "
140 StrArr$(3)="name "
150 StrArr$(4)="?"
160 DISP " Before calling binary."
170 DISP "------------------------"
180 FOR I=0 TO 4
190   DISP StrArr$(I);"/";
200 NEXT I
210 DISP @ DISP
```

[1] Due to the complexity of manipulating Series 500 FORTRAN string arrays and length headers, this topic is not discussed in this manual.

```
220 !
230 LOADBIN "fas"
240 CALLBIN "f_init" !  Initialization routine.
250 CALLBIN "farrays" (StrArr$())
260 !
270 DISP " After calling binary."
280 DISP "-------------------------"
290 FOR I=0 TO 4
300   DISP StrArr$(I);"/";
310 NEXT I
320 DISP @ DISP
330 !
340 END
```

Here is an example of a Series 200/300 FORTRAN binary that would work with the preceding `CALLBIN` statement.

```
      SUBROUTINE farrays(StringArray)
C
      CHARACTER *5 StringArray(5), TempArray(5)
C
C      First copy the passed array.
      TempArray(1) = StringArray(1)
      TempArray(2) = StringArray(2)
      TempArray(3) = StringArray(3)
      TempArray(4) = StringArray(4)
      TempArray(5) = StringArray(5)

C      Now rearrange elements.
      StringArray(1) = TempArray(3)
      StringArray(2) = TempArray(4)
      StringArray(3) = TempArray(2)
      StringArray(4) = TempArray(1)

      RETURN
      END
```

Here are the results of running the program.

```
 Before calling binary.
-------------------------
What /is   /your /name /?/

 After calling binary.
-------------------------
your /name /is   /What /?/
```

Note once again that the FORTRAN program modifies some of the string variables' lengths; however, the BASIC program string length is **not** modified. (See the preceding section for a work-around.)

Using Files with FORTRAN Binaries

BASIC programs and FORTRAN binaries can also communicate via BASIC type `text/data` files—HP-UX "ASCII" files. The main restriction is that BASIC and FORTRAN use the file exclusive of one another. That is, if BASIC opens and accesses a file, then it should close the file before the binary attempts to access the file.

Here is an example of a BASIC program opening a text file, putting some words into it, then closing it before calling a binary.

```
100 ASSIGN 11 TO "text_file" !  Open text file (BASIC will
110 !                            create it, if non-existent).
120 OUTPUT 11 USING "K" ; "What " ! Write items into file.
130 OUTPUT 11 USING "K" ; "is   "
140 OUTPUT 11 USING "K" ; "your "
150 OUTPUT 11 USING "K" ; "name "
160 OUTPUT 11 USING "K" ; "?    "
170 !
180 DISP " Before calling binary."
190 DISP "--------------------------"
200 ASSIGN 11 TO "text_file" ! Reset file pointer.
210 FOR I=1 TO 5
220   ENTER 11 ; String$ !     Read string items from file.
230   DISP String$;"/";
240 NEXT I
250 DISP @ DISP
260 ASSIGN 11 TO "*" !         Close file.
270 !
280 LOADBIN "ftf"
290 CALLBIN "f_init" !  Initialization routine.
300 CALLBIN "ftf"
310 !
320 DISP " After calling binary."
330 DISP "--------------------------"
340 ASSIGN 11 TO "text_file" ! Reset file pointer.
350 FOR I=1 TO 5
360   ENTER 11 ; String$ !     Read string items from file.
370   DISP String$;"/";
380 NEXT I
390 DISP @ DISP
400 !
410 ASSIGN 11 TO "*" !         Close file.
420 SCRATCHBIN "ftf"
430 END
```

Here is an example FORTRAN binary that would work with the preceding BASIC program.

```
      SUBROUTINE ftf

      CHARACTER *5 String, StringArray(5),ArrayCopy(5)

C     Read the text file contents.
      OPEN(45,FILE='text_file',STATUS='OLD')
      DO 10 I=1,5
        READ(45,'(A5)') String
        StringArray(I)=String
        ArrayCopy(I)=StringArray(I)
   10 CONTINUE
      CLOSE(45)

C     Now rearrange words.
      StringArray(1)=ArrayCopy(3)
      StringArray(2)=ArrayCopy(4)
      StringArray(3)=ArrayCopy(2)
      StringArray(4)=ArrayCopy(1)

      OPEN(45,FILE='text_file',STATUS='OLD')
      DO 20 I=1,5
        WRITE(45,'(A5)') StringArray(I)
   20 CONTINUE
      CLOSE(45)

      RETURN
      END
```

Here are the results of the BASIC program calling the FORTRAN binary.

```
 Before calling binary.
-------------------------
What /is   /your /name /?    /

 After calling binary.
-------------------------
your /name /is   /What /?    /
```

Notes

Graphics 15

Graphics are a powerful means of presenting information. Computer graphics can be equally powerful but an extra step is required between the conception of the idea and the final image. This step is the construction of a mathematical model of the image within the computer.

Since computers only do what they are told, it is essential to have a complete knowledge of the commands that communicate between the real world and the computer's world. This knowledge is needed to create the model within the computer's memory and to understand the resulting image of the model.

Chapter Contents

Tasks/Topics	Page
Example programs	15-2
Determining your device's capabilities	15-6
Specifying graphics devices	15-9
Limits and coordinates	15-12
Graphics input	15-52
Pens	15-54
Moving the pen and drawing	15-56
Labeling	15-65
Plotting and reading pixels	15- 69
Storing and retrieving raster images	15-73
Using plotters	15-75

Example Graphics Programs

This section shows two examples of graphics programs—one for graphics output, and one for both graphics input and output.

NOTE

These examples are intended to briefly show you the elements of graphics programming. They should allow you to begin using this graphics system as quickly as possible.

If you want more information on any particular statement, the tables following the programs list which section of the manual elaborates on the statement.

Example of Graphics Output (Plotting)

Here is an example of a graphics image created by a Technical BASIC program.

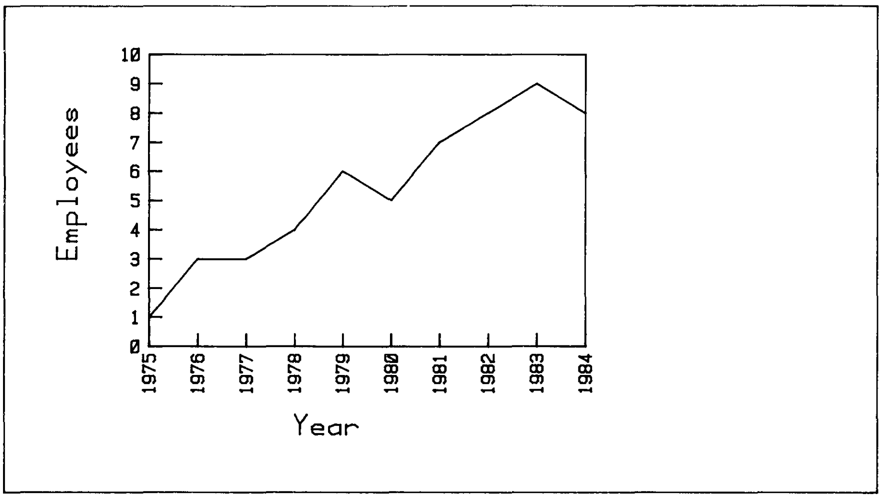


Figure 15-1. Example of Plotting

Here is the program that created the picture.

```
100 CLEAR !        Clear alpha raster.
110 PLOTTER IS 1 ! Choose plotter (and initialize it).
120 GCLEAR !       Clear any existing graphics.
125 !
130 FRAME !                Draw line around plotting bounds.
140 LOCATE 30,100,30,90 ! Define smaller plotting area.
150 FRAME !                Draw new bounds.
160 !
170 SCALE 1975,1984,0,10 !   Scaling maps into LOCATE bounds.
180 LAXES 1,1,1975,0,1,1,5 ! Create labeled axes.
190 !
200 MOVE 1978,-3 !    Move pen (before labeling x axis).
210 CSIZE 6,0.5 !     Use taller, narrower characters.
220 LABEL "Year"
230 !
240 MOVE 1973,3 !     Move pen (before labeling y axis).
250 DEG @ LDIR 90 !   Label direction is up (90 degrees).
260 LABEL "Employees"
270 !
280 DIM Employees(9) !  Array with 10 points (OPTION BASE 0).
290 DATA 1,3,3,4,6,5,7,8,9,8
300 FOR Year=1975 TO 1984
310   READ Employees(Year-1975) !      Read DATA item.
320   DRAW Year,Employees(Year-1975) ! Draw to new x,y location.
330 NEXT Year
340 !
350 END
```

Program Line	For Further Explanation, See:	Page
`110 PLOTTER IS`	"Specifying Graphics Devices" section	15-9
`120 GCLEAR`	*BASIC Reference Manual*	

`130 FRAME`
`140 LOCATE`
`150 FRAME`
`170 SCALE`
	"Limits and Coordinates" section	15-12

`180 LAXES`
`210 CSIZE`
`220 LABEL`
`250 DEG @ LDIR`
`260 LABEL`
	"Labeling" section	15-65

`200 MOVE`
`240 MOVE`
`320 DRAW`
	"Moving the Pen and Drawing" section	15-56

Example of Graphics Input

The following example uses a graphics input device to pick points, which are then connected by lines on the plotting device. (Note that the 'digitize button' is the Return key if your input device is your keyboard.) Here is an example of what you can draw using the program.

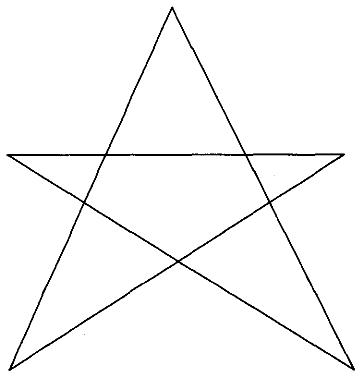


Figure 15-2. Simple Example of Digitizing

```
100 CLEAR !         Clear alpha raster.
110 PLOTTER IS 1 !  Specify CRT as plotter.
120 !
130 DISP "1. Move cursor with input device."
140 DISP "2. Press 'digitize button' to pick point."
150 DISP "3. Repeat 5 more times."
160 DISP
170 !
180 ! Pick first point.
190 DIGITIZE Xold,Yold
200 !
210 FOR Point=1 TO 5 !  Connect point with preceding one.
220   DIGITIZE Xnew,Ynew
230   MOVE Xold,Yold !  Move pen to last point.
240   DRAW Xnew,Ynew !  Draw line to this point.
250   Xold=Xnew !       Save last point.
260   Yold=Ynew
270 NEXT Point
280 DISP "Finished."
290 !
300 END
```

Program Line	For Further Explanation, See:	Page
`110 PLOTTER IS`	"Specifying Graphics Devices" section	15-9

`190 DIGITIZE`
`220 DIGITIZE`
	"Graphics Input" section	15-52

`230 MOVE`
`240 DRAW`
	"Moving the Pen and Drawing" section	15-56

Determining Your Device's Capabilities

These are the three basic capabilities that your graphics device(s) may have[1]:

- **Output (or "plotting"):** operations such as moving the pen, drawing lines, labeling, and setting up coordinate systems.
- **Input (or "digitizing"):** operations in which you move an "input locator," whose "x,y" coordinates the computer can determine.
- **Block read/write (or "bit block transfer"):** operations in which individual or groups of pixels (picture elements) on a raster display are read or written.

To determine the capabilities of your particular device(s), see the discussion of the **ASSIGN** statement in the *HP-UX Technical BASIC Reference Manual.*

NOTE

Many graphics devices have both "input" and "output" capabilities. For instance, the graphics raster of an HP 2627 Graphics Terminal is an output device, and the keyboard's cursor and Return keys are the input device.

List of Output (Plotting) Capabilities

There are several categories of graphics output operations:

- Graphics device selection.
- Graphics display control.
- Graphics boundaries and scaling.
- Moving the pen and drawing.
- Labeling.

Operations in each of these categories are listed below.

[1] In order to perform any graphics operations, you **must** have installed the *Starbase Graphics Library* on your HP-UX system. See the Starbase manuals for installation instructions.

Graphics Device Selection

`PLOTTER IS`	Specifies the plotting device (and input device).

Graphics Display Control

`CLEAR`	Clears the alpha display (and also the graphics display, on devices that use the same raster for both alpha and graphics).
`DUMP GRAPHICS`	Outputs the graphics display to the system printer.
`GCLEAR`	Clears all or portions of the graphics display.
`GRAPHICS`	Turns on the graphics display (if not already on; this is useful only on displays with *separate* alpha and graphics rasters).

Graphics Limits and Coordinates

`CLIP`	Specifies plotting boundaries in current scale units.
`LIMIT`	Specifies graphics limits in millimeter units.
`LOCATE`	Specifies the plotting boundaries in GU's.
`MSCALE`	Scales the plotting area in millimeter user units.
`RATIO`	Returns the ratio of the graphics limits—horizontal/vertical.
`SCALE`	Scales the plotting area by the specified user units.
`SETGU`	Sets the system to graphics units mode.
`SETUU`	Sets the system to user units mode.
`SHOW`	Scales the plotting area with equal x and y user units.
`UNCLIP`	Sets the plotting boundaries equal to the graphics limits.

Moving the Pen and Drawing

`AXES`	Plots x and y axes.
`DRAW`	Draws a line to the specified point.
`FRAME`	Draws a frame around the plotting area.
`GRID`	Draws grid lines.
`IDRAW`	Draws a line incrementally to the specified point.
`IMOVE`	Lifts the pen and moves it incrementally to the specified point.
`IPLOT`	Moves the pen incrementally to the specified point with pen control.
`LAXES`	Draws and labels x and y axes.

LGRID	Draws and labels a grid.
LINE TYPE	Specifies the line type used for lines, axes, and grids.
MOVE	Lifts the pen and moves it to the specified point.
PDIR	Establishes plotting direction for relative and incremental plotting.
PEN	Specifies the pen number.
PENUP	Lifts the pen.
PLOT	Moves the pen to the specified point with pen control.
RPLOT	Moves the pen with pen control to a point specified relative to a moveable origin.
WHERE	Reads the location of the "logical" pen.
XAXIS	Draws an x axis.
YAXIS	Draws a y axis.

Graphics Labeling

CSIZE	Establishes character size and shape for labels.
FXD	Specifies the format of floating-point numeric data items in labels for LAXES and LGRID.
LABEL	Plots a label at the current pen position.
LAXES	Draw and labels x and y axes.
LDIR	Specifies label direction.
LGRID	Draws and labels a grid.
LORG	Defines the position of labels relative to the current pen position.

List of Input Capabilities

CURSOR	Reads the location and status of the input locator.
DIGITIZE	Suspends program execution until the input locator's position is entered.

List of Block Read/Write Capabilities

BPLOT	Plots individual (or groups of) pixels on a graphics-raster display.
BREAD	Reads individual (or groups of) pixels on a graphics-raster display.
Complementing pens	These can be used to *exclusive OR* the current pen and existing pixels.

Specifying Graphics Devices

The PLOTTER IS statement specifies the graphics output and input device(s). This statement should precede all other graphic statements, as this statement initializes the graphics system by setting up default conditions. Once this statement is successfully executed, the graphics system is ready for use.

The following statement *usually* works to specify your console or terminal display:

```
PLOTTER IS 1
```

There are two possible outcomes of executing this statement:

- If **no error** is reported, execute the following statement:

  ```
  FRAME
  ```

  If lines are drawn around your plotting area, then the device has been selected and properly initialized. (If this is the case, skip to the section called "Explicitly Specifying Separate Input and Output Devices"—about 2 pages ahead.)

  If the FRAME statement does nothing, then you will need to use the procedure described in the next section.

- If an **error** is reported when you execute the PLOTTER IS 1 statement, you will need to use the following procedure to explicitly specify a plotting device.

Explicitly Specifying a Plotting Device

Assign a *device selector* to your plotting device. You can use any integer in the range 3 through 10, as long as it is not currently assigned (with an ASSIGN statement). Assign it to the appropriate *Starbase driver type* and *special (device) file.*

```
ASSIGN device selector TO "Starbase type,device file"
```

The *Starbase type* identifies the type of Starbase (graphics library) driver that is to be used with this resource. (A driver is a program that is used by the system to communicate with a particular device or interface.)

The *device file* parameter is the actual name of an HP-UX file that the System Administrator associated with an interface or device (using the HP-UX mknod, "make node," command). This file is assumed to be in the /dev directory if a pathname is not specified.

Here are some examples:

```
ASSIGN 5 TO "hp2623,tty"
PLOTTER IS 5

ASSIGN 6 TO "hp262x,/dev/tty_name"
PLOTTER IS 5

ASSIGN 7 TO "hp300h,/dev/device_file"
PLOTTER IS 7

ASSIGN 9 TO "hpwindow9837,/dev/screen/TBgraph1"
PLOTTER IS 9
```

NOTE

Once you have assigned a device selector to a Starbase resource, you must cancel the assignment (such as `ASSIGN 5 TO "*"`) before attempting to make another assignment using the *same* device selector.

Many output devices also have input capabilities; for instance, most terminals have input capabilities through the use of their graphics-cursor positioning keys.

- If your device *does* have input capabilities (and you either don't have or don't want to use a separate input device), then you need not read the following section. Skip to the subsequent "Coordinate Systems" section, or to the section that will answer questions about your particular graphics tasks.
- If you want or need to use a separate input device, then read the following section.

Explicitly Specifying Separate Input and Output Devices

When you have a plotting device *without* input capabilities, such as an HP 2625 Terminal or an HP 9837 Console display, or if you want to use separate input and output devices, you will need to use the ASSIGN and PLOTTER IS statements to specify the input device. (See the *Technical BASIC Reference Manual* for a complete list of devices, capabilities, and examples of assigning device selectors).

Example 1: A Display and an HIL Locator

The output device, specified first, is an HP 9837A display (hp9837,/dev/hp9837dev_file). The input device is the HP Human Interface Link (HIL) device at HIL address 2 (hil,hil2).

```
100  ASSIGN 5 TO "hp9837,hp9837dev_file; hil,hil2"
110  PLOTTER IS 5
```

NOTE

If an HP Human Interface Loop (HIL) input device is currently assigned, then you must first cancel the assignment (for instance, ASSIGN 5 TO "*") before making a new assignment.

Example 2: A Graphics Window and an HIL Locator

The following output device is an HP Windows/9000 graphics window (hpwindow98700, hp98700). The input device is the HP Human Interface Link (HIL) device at HIL address 3 (hil,hil3).

```
100  ASSIGN 6 TO "hpwindow98700,/dev/screen/window1; hil,hil3"
110  PLOTTER IS 6
```

Example 3: A Plotter and an HP-IB Tablet

The following output device is an HP 7475A plotter (hpgl,/dev/hp7475A). The input device is an HP 9111 Graphics Tablet (hpgl,hp9111A) connected to the computer through an HP-IB interface.

```
100  ASSIGN 6 TO "hpgl,hp7475A; hpgl,hp9111A"
110  PLOTTER IS 6
```

NOTE

With hpgl Starbase types, such as HP-IB plotters and graphics tablets, the corresponding nodes must be "auto-addressed;" that is, there must be primary addressing in the minor number. See ASSIGN in the *Technical BASIC Reference Manual* for further details.

Limits and Coordinates

Since you create graphics by telling the computer where to draw points and lines, the drawing area must have a coordinate system that allows you to specify the locations of these points and lines. With Technical BASIC, there are several different methods available for setting up limits and a coordinate system.

Overview

The first example at the beginning of the chapter set some bounds within which plotting was to take place, and then set up a useful coordinate system within those bounds. This section goes further into the topic of limits and how various coordinate systems can be set up within these limits.

Physical Limits

The raster display and all other plotting devices have physical limits which **define the maximum size of graphics image** that can be produced on the device. For example, you cannot produce an image on a graphics display screen that is larger than its graphics raster. Similarly, the physical limits of a plotter determine the maximum size of drawing that it can produce.

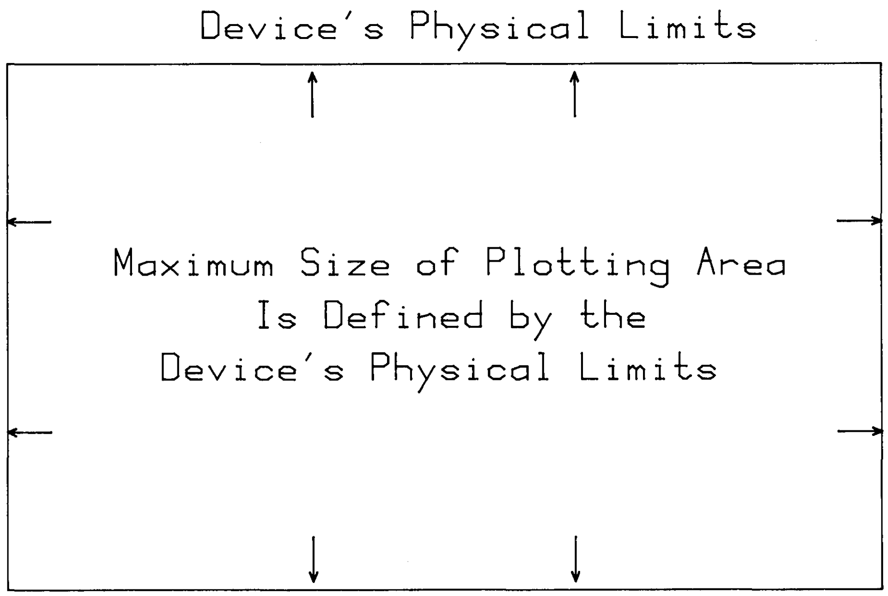


Figure 15-3. Physical Limits on a Graphics Output Device

Graphics Limits

Within the physical limits of a device, you can choose the location of the graphics output by setting the graphics limits. The graphics limits are the boundaries for all[1] graphics output.

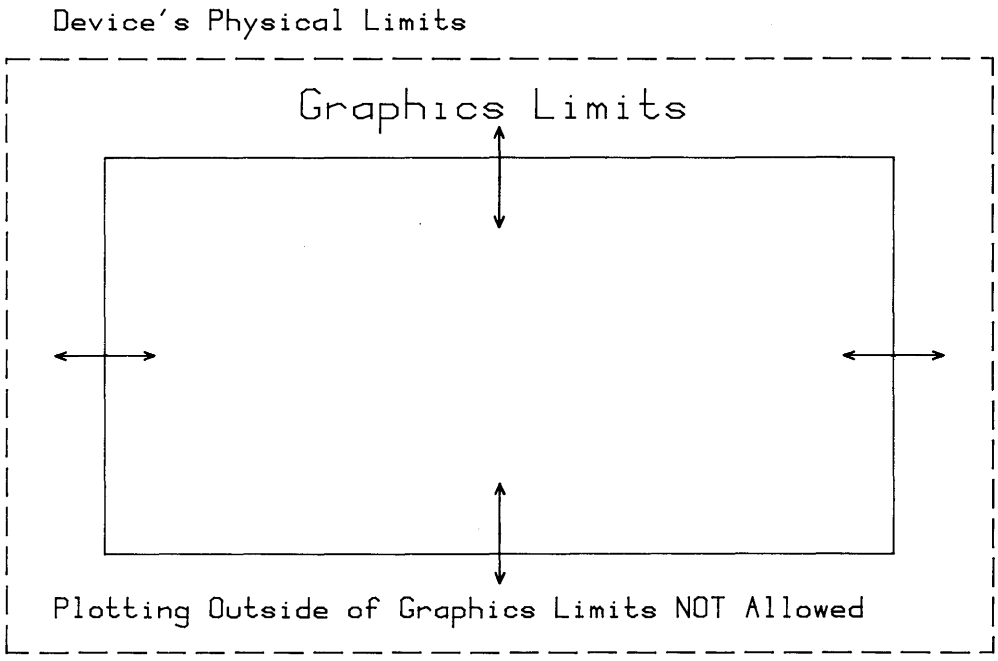


Figure 15-4. Graphics Limits Can Be Moved

The graphics limits are set to their default when the BASIC system is entered[2].

[1] The only exceptions to this statement are the byte-plotting operations performed on a graphics raster, which can be performed outside of the current graphics limits (but not outside the physical limits).

[2] The default graphics limits for your console's or terminal's graphics raster are described in the device's documentation. The default graphics limits vary for different external plotters, but are generally close to the physical limits of the device. Refer to the documentation accompanying the plotter for additional information regarding physical and default graphics limits.

Scaling Maps into the Graphics Limits

When you want to move the pen, you specify the coordinates to which you want the pen to move. For instance, this statement tells the pen to move to the coordinate 50,50.

```
MOVE 50,50
```

The *physical location* to which the pen moves depends on the coordinate system currently set up for the device. As an example, this sequence of statements moves the graphics limits, sets up a user units (UU's) coordinate system which maps into the graphics limits, and shows the coordinate system.

```
100  LOCATE 20,120,20,80 !        Sets plotting bounds (in GU's).
110  SCALE 0,140,0,100 !          Sets up UU coordinates.
120  LAXES 20,20,0,0,20,20,300 ! Draws and labels axes.
```

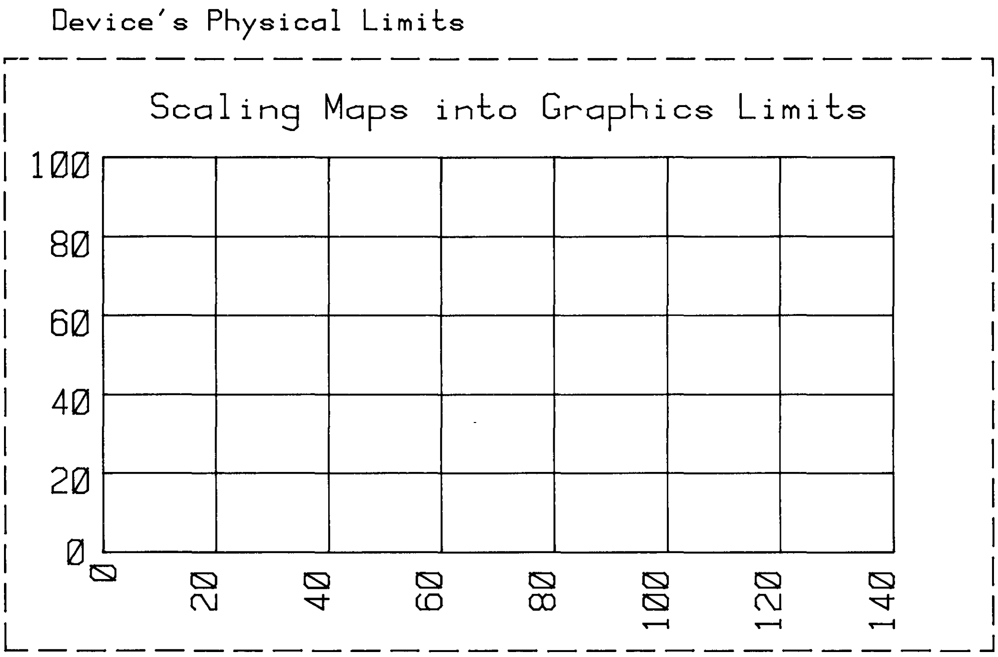


Figure 15-5. Scaling maps into Graphics Limits

The X coordinate of the left graphic limit is 0, while the X coordinate of the right limit is 140. The Y coordinates of the lower and upper graphics limits are 0 and 100, respectively.

The length of 1 user unit in the X direction is determined by the difference in X coordinates of the graphics limits (here 140) divided by the physical distance between right and left graphics limits (depends on the plotting device). The length of 1 user unit in the Y direction is determined similarly.

This is only a brief look at scaling. However, it does show how the coordinate system is mapped onto the physical plotting device. More details of scaling methods are shown in the next section. Details of changing the graphics limits are described in "Moving the Limits" section.

A Closer Look at Coordinate Systems

Once the plotting area is defined, either by default or by specifying the graphics limits, the scale can be chosen to suit your particular graphics application. You can use the default scale—graphics units (GU's)—or you can specify your own scale—user units (UU's). If you do not specify your own units, the BASIC system sets UU's equal to GU's.

The Default Coordinates: Graphics Units (GU's)

When the graphics system is initialized (by PLOTTER IS), the default scale is measured in Graphic Units. The origin (location 0,0) is in the lower left corner of the graphics raster. The shorter side of the raster is 100 GU's in length. The number of GU's in the longer side is determined by the aspect ratio: width/height. For instance, if the screen is exactly two times as wide as it is high, then its aspect ratio is 2. Thus the X coordinate of the right bound is 200, while the Y coordinate of the upper bound is 100.

This example shows the default (GU) scaling for a graphics raster which has an aspect ratio of a little greater than 2.

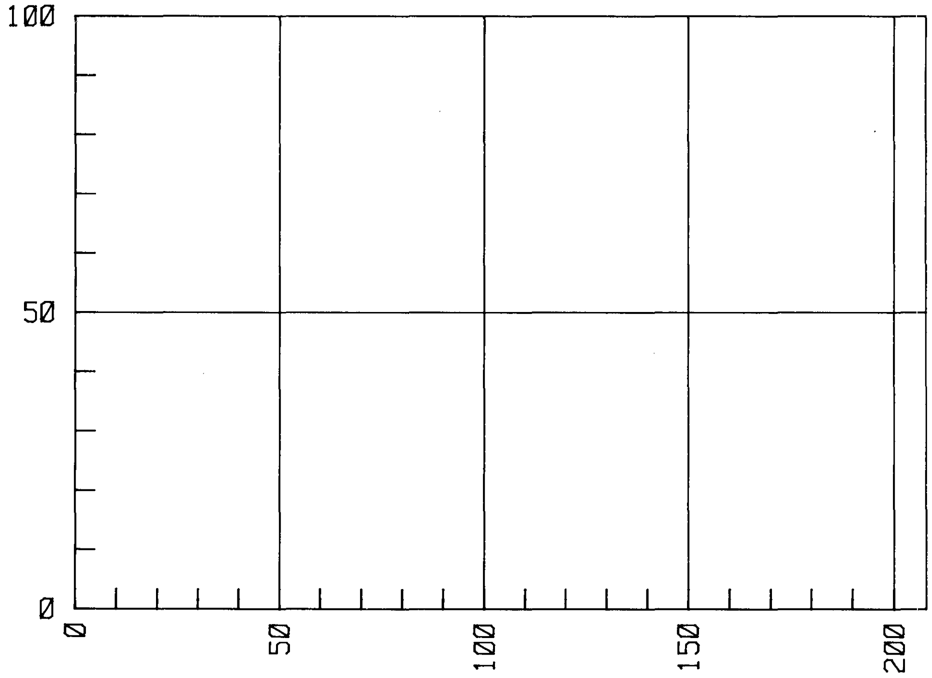


Figure 15-6. Example of Default Graphics Units (GU) Mapping

Graphics Units Mapping

The graphics units scale maps onto the area defined by the graphics limits. When the graphics limits change, the size of the graphics units scale also changes. For example, this statement moves the graphics limits to form a 50-by-70-millimeter plotting area:

```
LIMIT 10,60,0,70
```

The graphics unit scale now maps onto this plotting area. The length of one GU is equal to 1/100 (one percent) of the length of the shortest side of the area bounded by the graphics limits. The length of the longest side of the plotting area is again something greater than 100, depending on the width/height aspect ratio.

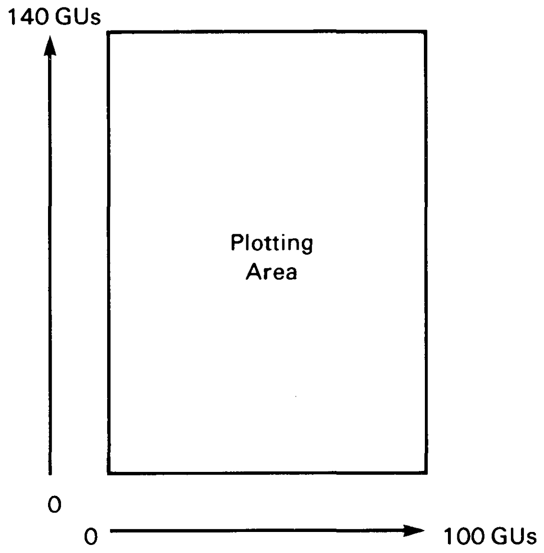


Figure 15-7. Graphics Units (GU) Map into LIMIT Bounds

The graphics units scale provides a simple method of scaling the plotting area on a percentage basis, regardless of the size of the plotting area.

The following program draws a line from point (0,0), in GUs, to the opposite corner of the plotting area. Enter the graphics limits from the keyboard; the RATIO function is used to compute the length in GUs of the longest side of the plotting area.

```
100 ! *** Graphics Units ***
110 PLOTTER IS 1 !                    The display is the plotter.
120 GCLEAR
130 DISP "Enter LIMIT parameter: xmin,xmax,ymin,ymax"
140 INPUT xmin,xmax,ymin,ymax
150 LIMIT xmin,xmax,ymin,ymax !       Specifies graphics limits.
160 DISP "RATIO = ",RATIO !           Displays current RATIO.
170 WAIT 2000
180 !
190 GCLEAR !                          Clears the graphics area.
200 FRAME !                           Frames plotting area.
210 MOVE 0,0 !                       Moves the pen to lower-left
220 !                                corner.
230 Xmax=100*MAX(1,RATIO) !         Maximum x value in GUs.
240 Ymax=100*MAX(1,1/RATIO) !       Maximum y value in GUs.
250 DRAW Xmax,Ymax !                 Draws a line to upper-right
260 !                                corner.
270 END
```

Execute the above program and enter the following data when prompted to do so.

```
10,110,5,55
```

An alpha display of the RATIO is given as the first part of the result.

```
RATIO = 2
```

The final part of the result from executing the program is this graphics display:

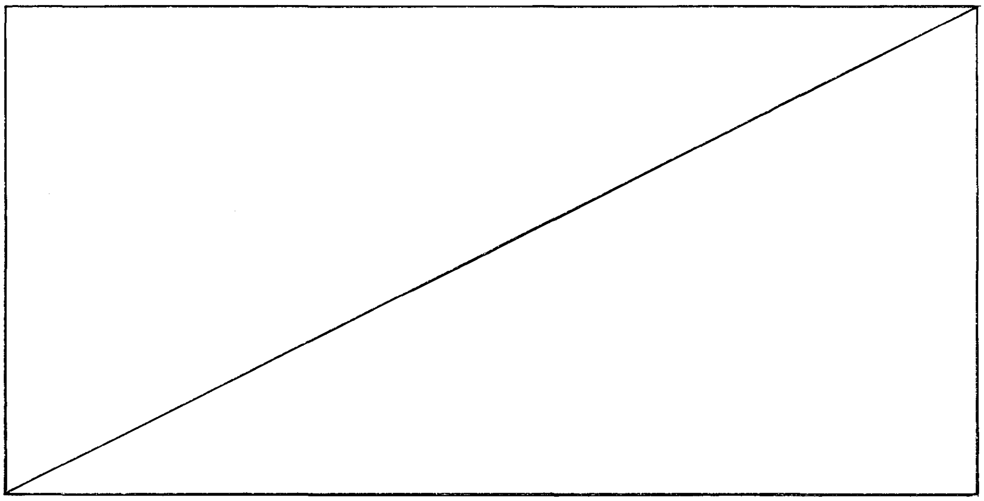


Figure 15-8. Using RATIO in GU Coordinate Calculation

Axes and Grids

The `AXES` statement can be used to draw axes and to put tick marks on the axes. The following example shows the use of all of the parameters, interpreted as GU's, available with the `AXES` statement:

```
AXES XtickSpc,YtickSpc,XLocYAxis,YLocXAxis,Xmajor,Ymajor,Size
```

The `XtickSpc` and `YtickSpc` parameters specify the number of units between the tick marks.

`XLocYAxis` and `YLocXAxis` specify the location of the intersection of the axes: `XLocYAxis` is the X location at which the Y axis crosses the X axis, and `YLocXAxis` is the Y location at which the X axis crosses the Y axis. If these parameters are not specified, the default intersection is:

- the lower, left corner (when tick spacing is not specified)
- 0, 0 (when tick-spacing is specified)

`Xmajor` and `Ymajor` specify which ticks are to be "major" (full-size) ticks; all other ticks will be "minor" (half-size) ticks. For example, if `Xmajor` is set to 4, then every fourth tick on the X axis will be a major tick.

Tick length is determined by the `Size` parameter. It specifies the size, in GU's, of the major ticks; minor ticks are always half the length of major ticks. If it is not specified, major ticks have a default length of 2 GU's, and minor ticks have a default length of 1 GU.

The following program shows two examples of axes.

```
100 PLOTTER IS 1
110 AXES 10,10 !       10 units between the ticks with origin at (0,0).
120 AXES 20,20,50,30 ! 20 units between the ticks with origin at (50,30).
130 END
```

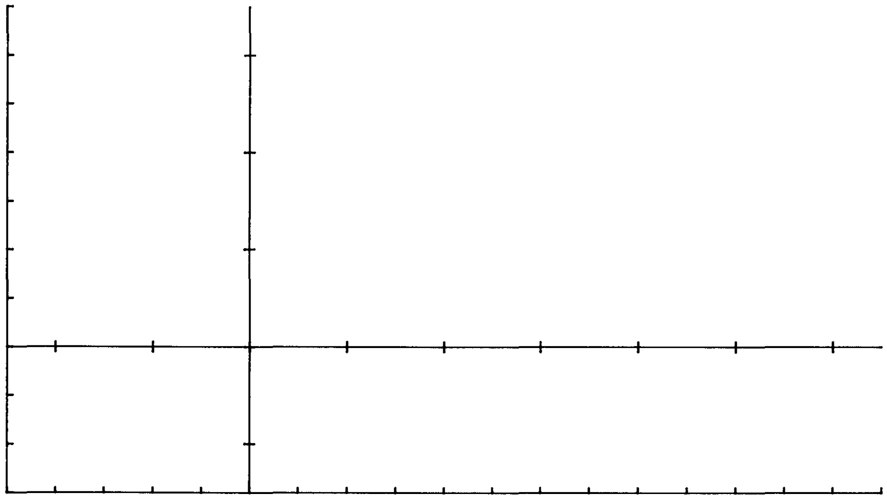


Figure 15-9. Using AXES to Draw X and Y Axes

The `AXES` statement has a related statement: `GRID`. This statement is best thought of as a pair of axes with very long tick marks. `GRID` uses the same parameters as `AXES`, except that the `Size` parameter specifies the minor tick length (since `GRID`'s "major ticks" span the plotting area).

```
GRID Xtick,Ytick,Xorigin,Yorigin,Xmajor,Ymajor,Size
```

Here are examples of using GRID in a program.

```
100 PLOTTER IS 1
110 GCLEAR
120 !
130 GRID 10,10 !          Grid with X and Y ticks 10 GU's apart.
140 WAIT 5000
150 !
160 GCLEAR
170 GRID 20,20,0,0,3,2,4 ! Ticks 10 GU's apart;
180 !                     origin at 0,0;
190 !                     X major tick every 3rd grid;
200 !                     Y major tick every 2nd grid;
210 !                     Tick length = 4 GU's.
```

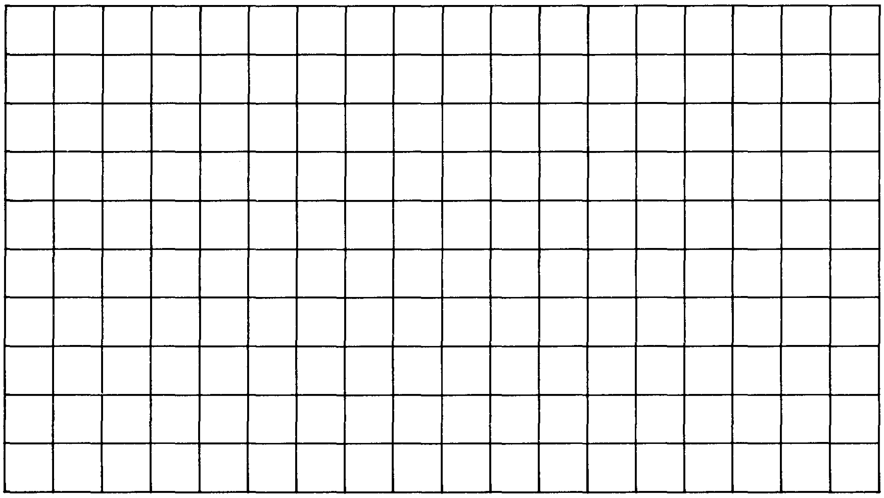


Figure 15-10a. Output of GRID 10,10

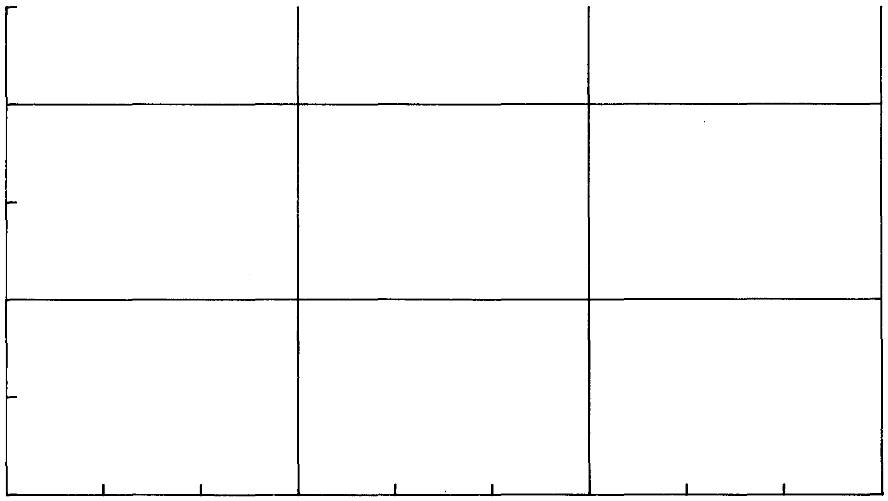


Figure 15-10b. Output of GRID 20,20,0,0,3,2,4

User Units

The first example in this chapter set up a more relevant plotting scale with this statement:

```
170 SCALE 1975,1984,0,10
```

The parameters define the coordinates of left, right, lower, and upper boundaries of the plotting area, respectively. The scaling units set up by this statement are known as User Units, or UU's.

Here is a SCALE statement that uses meaningful variable names to specify the parameters that set up a User Units coordinate system:

```
170 SCALE Left,Right,Bottom,Top
```

The next section describes UU's in greater detail.

User Units Scales

There are three scaling statements that allow you to specify the user units scale:

- SCALE—sets up scaling in user units (UU's)
- SHOW—like SCALE, but the units in X and Y directions are equal in length (isotropic scaling)
- MSCALE—sets up scaling in mm units

All three scaling statements specify the scale for the current plotting area (defined by the graphics limits) or by a LOCATE statement (which also specifies plotting boundaries).

Anisotropic Scaling (SCALE)

The SCALE statement defines the coordinates of the limits of the current plotting area. The syntax for scale is as follows:

```
SCALE x_min,x_max,y_min,y_max
```

The parameters can be numeric constants, variables, or expressions.

This program shows an example of setting up a user-units coordinate system:

```
100 ! *** Scale ***
110 PLOTTER IS 1 !                The display is the plotter.
120 GCLEAR !                      Clears the graphics display.
130 !
140 SCALE -2,2,-4,4 !             Specifies UU scale.
150 GRID 1,1,0,0 !                Draws a grid with  1 UU spacing.
160 !
170 DEG !                         Sets degrees mode.
180 MOVE 1,0 !                    Moves to the start of the ellipse.
190 FOR Degrees=0 TO 360 STEP 10 !  10 degree increments.
200   DRAW COS(Degrees),SIN(Degrees) ! Draws in UU's.
210 NEXT Degrees
220 END
```

The following graphics display is the result of executing the above program.

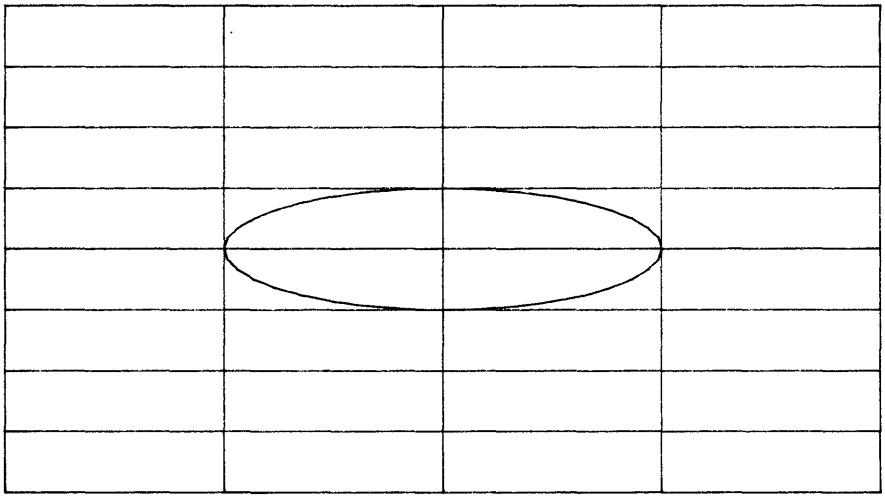


Figure 15-11. SCALE Sets Up UU Scaling (Shown by GRID)

The SCALE statement specifies user units independently in the X and Y directions.

Isotropic Scaling (SHOW)

The SHOW statement specifies user units for a plotting device such that one unit on the X axis is the same length as one unit on the Y axis (isotropic scaling). Thus, the plotting area is scaled with unit squares. The SHOW statement parameters specify the *minimum* number of units to be mapped onto the current plotting area. If necessary, units are added to a dimension to scale the plotting area isotropically (an example is provided subsequently).

The syntax for the SHOW statement is as follows:

```
SHOW x_min,x_max,y_min,y_max
```

To use the SHOW statement, replace the SCALE statement in the previous example with the SHOW statement. Because of equal unit scaling, the figure drawn is now a circle instead of an ellipse. Line 140 of your program should look like this:

```
140 SHOW -2,2,-4,4
```

and your changed program should be as follows:

```
100 ! *** Scale ***
110 PLOTTER IS 1 !             The display is the plotter.
120 GCLEAR !                   Clears the graphics display.
130 !
140 SHOW -2,2,-4,4 ! <<<------- Specifies 'isotropic' UU scale.
150 GRID 1,1,0,0 !             Draws a grid with  1 UU spacing.
160 !
170 DEG !                      Sets degrees mode.
180 MOVE 1,0 !                 Moves to the start of the ellipse.
190 FOR Degrees=0 TO 360 STEP 10 !  10 degree increments.
200   DRAW COS(Degrees),SIN(Degrees) ! Draws in UU's.
210 NEXT Degrees
220 END
```

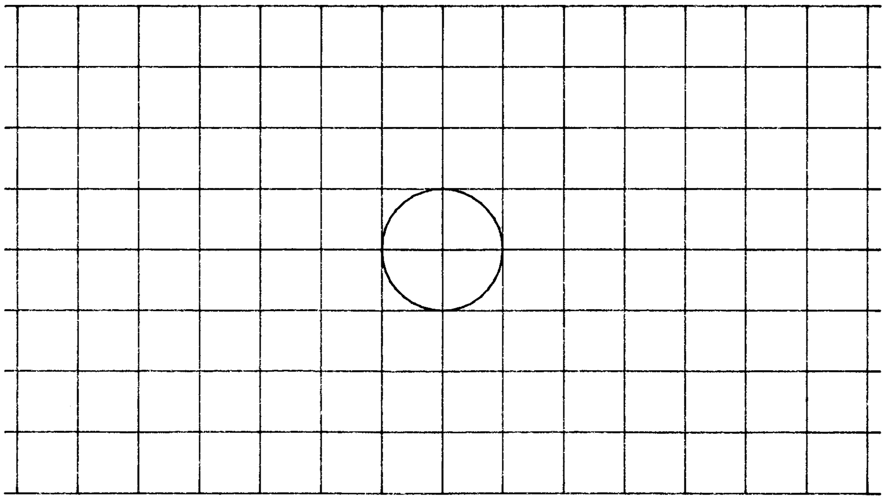


Figure 15-12. SHOW Sets Up Isotropic Scaling

The SHOW statement sets up UU's such that the coordinate system is as large as possible and is centered within the graphics limits (or within the plotting boundaries, if specified). For example, if the LIMIT rectangle is twice as wide as it is high (for example, LIMIT 0,100,0,50), then SHOW -1,1,-1,1 is equivalent to SCALE -2,2,-1,1.

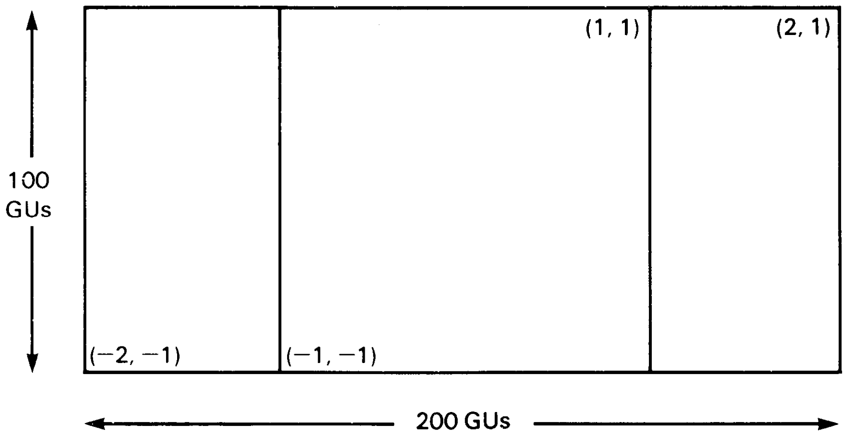


Figure 15-13. SHOW Sets Up as Large a Coordinate System as Possible

Millimetre Scale (MSCALE)

The MSCALE statement sets millimetres as user units and specifies the location of the origin. MSCALE is useful when correspondence between an image and a physical object is desirable, as in drafting applications. The accuracy of the scale depends *entirely* on the graphics device in use.

The MSCALE statement parameters are different than parameters in the SCALE and SHOW statements.

```
MSCALE x_offset,y_offset
```

MSCALE specifies user units equal to millimetres, and offsets the origin (0,0) in millimetre spacing, from the lower-left graphics limits corner by the specified distance, in millimetres. The MSCALE parameters can be numeric constants, variables, or expressions.

For example, the following statement specifies that 1 user unit equals 1 mm; the origin is offset 10 mm to the right and 15 mm up from the lower-left corner of the plotting area.

```
MSCALE 10,15
```

Like SCALE and SHOW, the MSCALE statement must follow operations that set or move the graphics limits or the LOCATE plotting boundaries in order to map the user units scale onto the current plotting area. MSCALE also works after default conditions have been restored by PLOTTER IS.

The following program uses the MSCALE statement to draw a metric ruler on the display.

```
100 ! *** Metric Ruler ***
110 PLOTTER IS 1
120 GCLEAR
130 FRAME
140 MSCALE 10,40 !          Specifies metric user units with
150 !                       10 mm x_offset and 40 mm y_offset.
160 CLIP 0,100,0,10 !       Clips plotting area in millimetres.
170 FRAME !                 Frames the plotting area (ruler).
180 AXES 2,10,0,10,5,10,5 ! Draws the ruler's metric scale.
190 MOVE 30,3
200 LABEL USING "K" ; "10 cm Metric Scale"
210 END
```

The results from executing this program are as shown in the following picture:

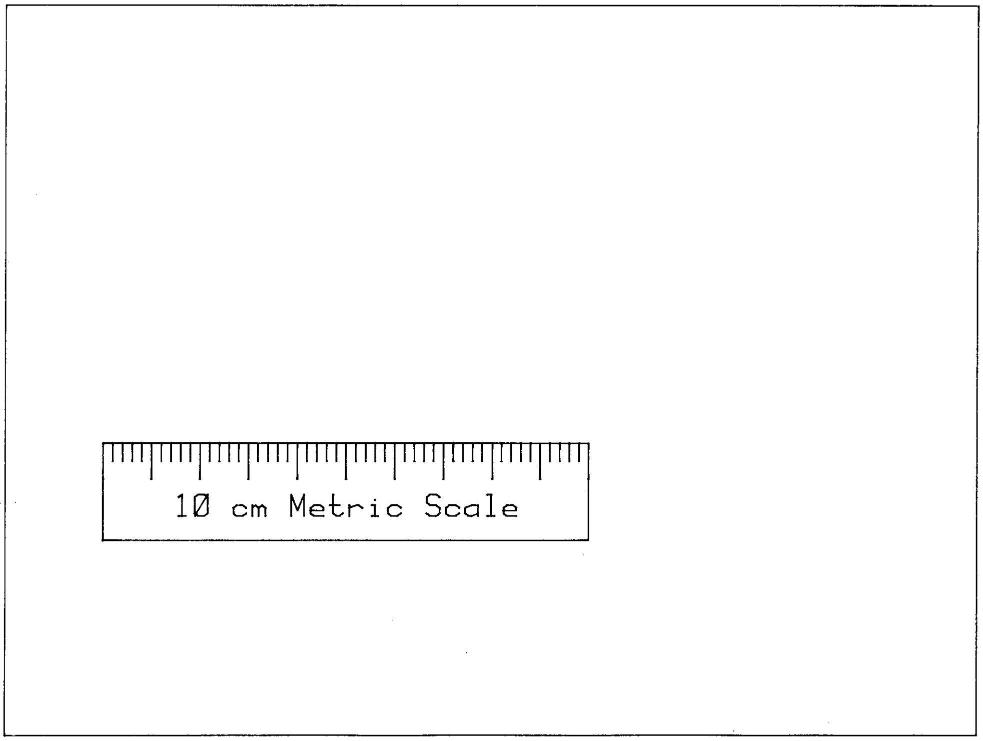


Figure 15-14. Example of MSCALE (Millimetre) Scaling

The specified `MSCALE` origin need not be located inside the graphics limits; you can plot, for example, in negative millimetre units by specifying the origin of your `MSCALE` beyond the maximum X and Y dimensions of the graphics limits.

The following program draws a metric grid; the MSCALE origin is offset to the upper-right corner of the plotting area.

```
100 ! *** Negative MSCALE ***
110 PLOTTER IS 1
120 GRAPHICS !                   Sets the display to graphics mode.
130 LIMIT 0,160,0,60 !           Specifies the graphics limits.
140 MSCALE 160,60 !              Specifies metric UUs and places
150 !                            the origin at the upper-right
160 !                            corner of the plotting area.
170 FRAME !                      Frames the plotting area.
180 GRID 2,2,0,0,10,10 !         Draws a metric grid with 10mm
190 !                            spacing and 2mm tic marks on the
200 !                            x and y axes.
210 END
```

Execution of the previous program results in the following display:

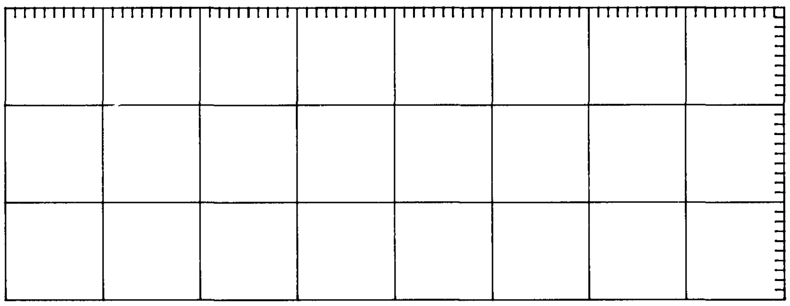


Figure 15-15. Example of MSCALE (Millimetre) Scaling

Changing Units: SETGU and SETUU

The two types of units used by the computer in plotting operations are graphics units (GUs) and user units (UUs). The current units mode refers to the type of units in use during plotting. At entry to the BASIC system, the computer is set to user units mode and the current user units scale is GUs, by default. However, you can switch modes at any time and access the current UU's and GU's scales by executing the mode change statements: `SETGU`, and `SETUU`.

The `SETGU` statement sets the system to graphics units mode, establishing GU's as the current scale. Executing `SETGU` does not disturb the current user units scale, and allows you to plot outside the plotting boundaries set by the `LOCATE` and `CLIP` statements (discussion of plotting boundaries appears later in this section). The `SETGU` statement is the only means by which the computer is set to graphics mode. Unless `SETGU` is executed, the computer plots according to the current user units scale as defined by `SCALE`, `SHOW`, `MSCALE`, or by default (GU's). The syntax for setting the graphics unit mode is:

```
   SETGU
```

`SETUU` sets user units (UUs) as the current units mode. User units mode is also set by the `SCALE`, `SHOW`, `MSCALE`, `LIMIT`, and `PLOTTER IS` statements, and by default. The syntax for setting the user units mode is as follows:

```
   SETUU
```

If the system is set to graphics units mode, you need to return it to user units mode in order for the plotting boundaries set by `LOCATE` or `CLIP` to be active. `SETGU` establishes the area bounded by the graphics limits as the current plotting area.

The following program makes use of both scales: UU's and GU's. The GU's scale is determined by the graphics limits, and is recalled by the SETGU statement.

```
100 PLOTTER IS 1 !      Sets to UU's (=GU's now).
110 GCLEAR
120 FRAME !             Show display limits.
130 CSIZE 3
140 !
150 LOCATE 40,100,30,80 ! Move plotting bounds (GU's).
160 FRAME
170 !
180 SCALE -20,20,-20,20 !  Scale in UU's.
190 MOVE 0,0
200 LABEL "    0,0 UU's"
210 DRAW 20,20
220 LABEL "20,20 UU's"
230 !
240 SETGU !      Change back to GU's.
250 MOVE 2,2 !
260 LABEL "    0,0 GU's"
270 MOVE 0,0
280 DRAW 20,20 !
290 LABEL "  20,20 GU's"
300 END
```

Execution of the previous program results in the following display:

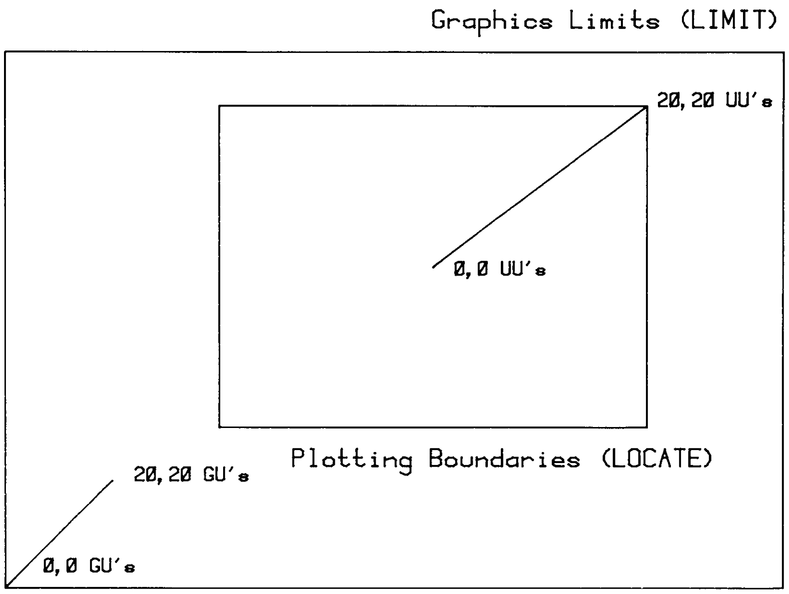


Figure 15-16. Changing from GU's to UU's (and Back)

Once a scaling statement is executed, the current user-defined scale is active until one of the following occur:

- A new scaling statement is executed (`SCALE`, `SHOW`, or `MSCALE`).
- The system is exited and re-entered, in which case UU's default to GU's.
- A `LIMIT` or `PLOTTER IS` statement is executed (UU's set equal to GU's).
- The system is switched to graphics units mode by executing `SETGU`.

Moving the Graphics Limits

A device's *physical limits* (such as "P1" and "P2" on plotters) are read by the BASIC system when it executes a `PLOTTER IS` statement. These limits are the default *plotting bounds*. You can move these plotting bounds with the `LIMIT`[1] statement.

```
LIMIT xmin,xmax,ymin,ymax
```

Since the `LIMIT` parameters are **in millimetres**, they specify the **absolute** locations of the graphics limits. The origin (0,0) is normally the lower-left physical limit of the plotting device. X coordinates increase as you move toward the right physical limit of the plotting area, while Y coordinates increase toward the top physical limit. `LIMIT` enables you to move the graphics limits anywhere within the physical limits of the plotting device.

The following program shows an example of default and user-defined graphics limits:

```
100 ! *** Limit ***
110 PLOTTER IS 1 !          Specifies the CRT as the plotter
120 !
130 GCLEAR !                Clears the graphics display.
140 LINE TYPE 5 !           Specifies the dashed line type.
150 FRAME !                 Frames the plotting area for
160 !                       reference.
170 LIMIT 90,90+80,70,70+40 ! Specifies an 80 mm X 40 mm plotting
180 !                       area that is offset from the display's
190 !                       lower-left physical bounds.
200 FRAME !                 Frames the specified plotting area.
210 END
```

[1] `LIMIT` does **not** change P1 and P2 on plotters. It does, however, restore default graphics conditions. See the *HP-UX Technical BASIC Reference Manual* for further details about graphics defaults.

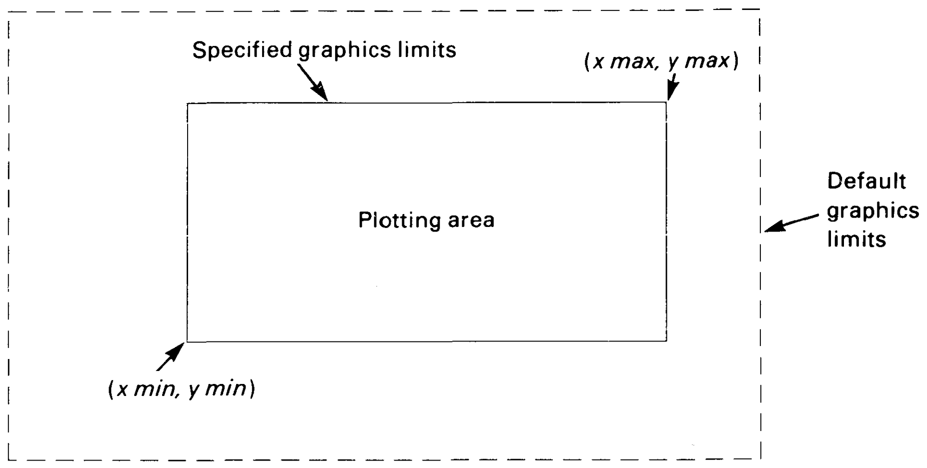


Figure 15-17. Default and User-Defined Graphics Limits

Scope of LIMIT Statements

As demonstrated by the program, the `LIMIT` statement overrides any previously set or default graphics limits. These graphics limits remain in effect until one of the following operations is performed:

- Another `LIMIT` statement is executed.
- A `PLOTTER IS` statement is executed.
- The BASIC system is exited and re-entered.

If you do not execute a `LIMIT` statement in a program and your plot turns out smaller than you expected, then the plotting device is probably using the graphics limits set by a previous `LIMIT` statement.

Range of Graphics Limit Parameters

The ranges of the `LIMIT` parameters are determined by the current `PLOTTER IS` device's physical limits. For the graphics raster of your console or terminal, the range of `LIMIT` parameters are supplied in the *Implementation Specifics* appendix for your particular Technical BASIC system. For external plotters, the range is given in the documentation supplied with the plotter.

If a `LIMIT` statement parameter is out-of-bounds, the system may return an error message and either ignore the statement or set the limits to the defaults.

Aspect Ratio

The current output device's aspect ratio (width/height) is returned by the following BASIC function.

```
RATIO
```

Here is how the `RATIO` is calculated:

`RATIO` = width / height = (Xmax-Xmin) / (Ymax-Ymin)

Thus `RATIO` has no units, since they cancel in the division of width by height.

`RATIO` can be used to determine the length of the longer side of the plotting area, since GU's are specifically chosen so that the shorter of the plotter's width or height is exactly 100 GU's long. If the height is shorter than the width, then this expression gives the plotting area's width (in GU's).

```
100*RATIO
```

If the width is shorter than the height (indicated by RATIO returning a value less than one), then this expression gives the plotting area's height (in GU's).

```
100/RATIO
```

The value of the `RATIO` function depends on the current graphics limits, which can be set by default, by `LIMIT`, or manually on the plotting device.

```
10 LIMIT 5,95,10,60 ! New graphics limits (in mm).
20 DISP RATIO !       Width/height=(95-5)/(60-10)=90/50.
30 END
```

The value returned by `RATIO` is:

```
1.8
```

The RATIO function is useful for determining the aspect ratio of the graphics limits before changing the size or location of the plotting area while maintaining the same aspect ratio. A sample program is given below.

```
100 ! *** RATIO ***
110 PLOTTER IS 1 !
120 LIMIT 0,460/2.84,0,192/2.84 ! Restore default limits.
130 GCLEAR
140 FRAME !                       Show default limits.
150 !
160 LIMIT 20,110,0,65 !           Move limits.
170 FRAME !                       Show   "
180 !
190 R=RATIO !                     Determine width/height.
200 LIMIT 5,R*(60-20)+5,20,60 !   Use RATIO in new limits.
210 FRAME !                       Show new limits.
220 !
230 LIMIT 70,100,10,(100-70)/R+10 ! Move limits.
240 FRAME !                         Show them.
```

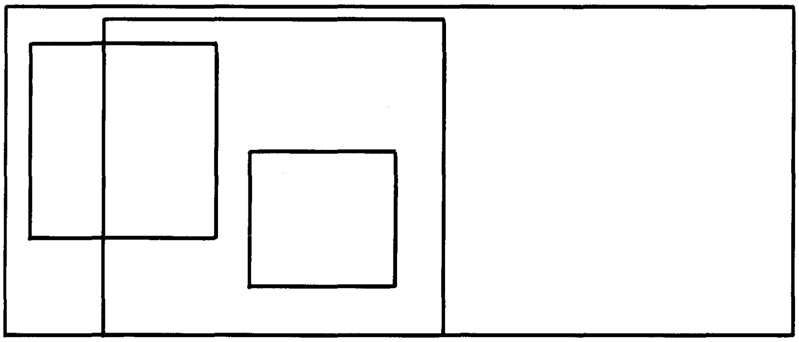


Figure 15-18. Maintaining Aspect Ratios When Setting New Limits

Plotting Boundaries

While plotting is restricted either to the default graphics limits or those specified by a `LIMIT` statement, the `LOCATE` and `CLIP` statements specify plotting boundaries. Like the graphics limits, plotting boundaries mark the limits of the plotting area. However:

- Plotting boundaries differ from graphics limits in that they represent **conditional limits**. Plotting outside the plotting boundaries is possible while the system is set to graphics units, GU, mode, or while labeling. (For instance, plotting boundaries can be set to reserve space within the graphics limits for labeling.)
- Plotting boundaries can be used to create windows for showing portions of a plot.

The diagram below shows different ways in which plotting boundaries can be set with respect to the graphics limits. Although the plotting boundaries can extend beyond the graphics limits, or for that matter, the physical limits of the plotting device, you can't plot or label outside the graphics limits.

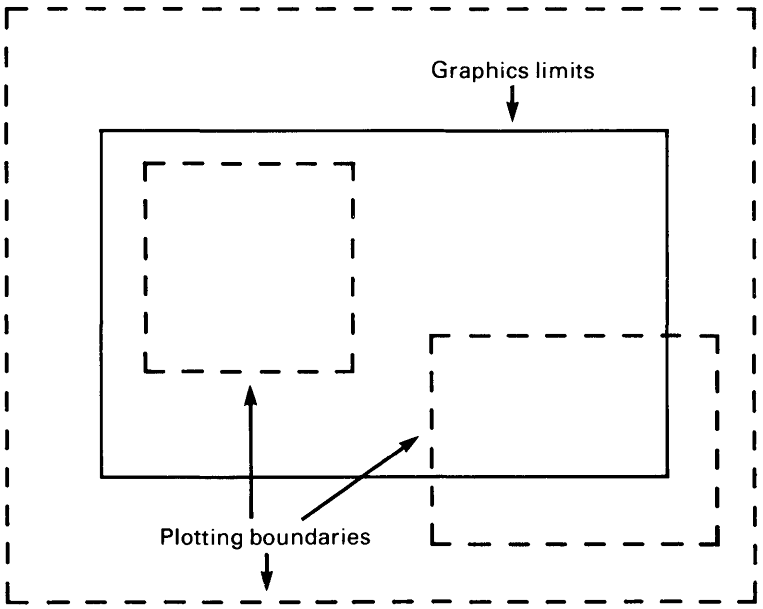


Figure 15-19. Plotting Boundaries and Graphics Limits

LOCATE Boundaries

The LOCATE statement enables you to relocate the plotting area within the graphics limits by specifying the plotting boundaries. This allows you to leave space for labels outside of the plotting area, but within the graphics limits. The parameters in the LOCATE statement are expressed in GU's. Thus, LOCATE defines the plotting boundaries as a percentage of the graphics limits. The syntax for the LOCATE statement is as follows:

```
LOCATE Xmin,Xmax,Ymin,Ymax
```

The first two parameters specify the left and right boundaries, and the last two parameters specify the lower and upper boundaries. The parameters can be numeric constants, variables, or expressions.

Like the LIMIT statement, LOCATE can be exchanged to reflect the plot across the x and y axes. However, note that the reflection does not begin until a scaling operation is performed (such as SCALE or SHOW). Refer to the section of this chapter called "Reflecting Images" for further details.

When the LOCATE statement is executed prior to a scaling statement (SCALE, SHOW, or MSCALE), the user units scale is mapped onto the area defined by LOCATE rather than the graphics limits.

The plotting boundaries specified by the LOCATE statement replace any previously defined plotting boundaries. In turn, the LOCATE-defined plotting boundaries are redefined by the CLIP statement. The LIMIT, UNCLIP, and PLOTTER IS statements default the plotting boundaries to the graphics limits. The plotting boundaries are also set to the graphics limits whenever display memory is reapportioned or the computer is turned on. When the computer is set to graphics units mode by executing SETGU, the graphics limits define the current plotting area. Executing SETUU restores the LOCATE or CLIP-defined plotting boundaries.

The following drawings show the available plotting area and the current scale in user units mode and graphics units mode for the given `LIMIT`, `LOCATE`, and `SCALE` statements. Labeling is allowed anywhere within the graphics limits, regardless of the current units mode. The graphics limits are drawn in solid lines; the plotting boundaries are drawn in dotted lines. The plotting area is the shaded portion.

```
LIMIT 0,120,0,60
LOCATE 100,300,50,150
SCALE 0,10,0,10
```

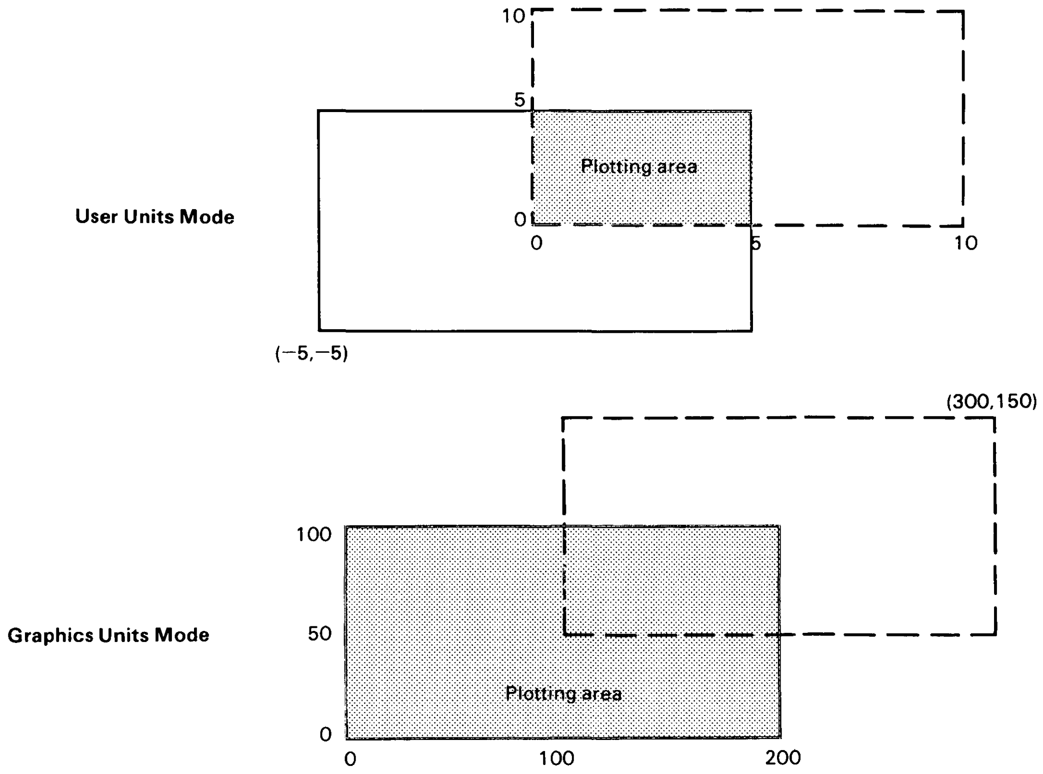


Figure 15-20a. LOCATE, UU's, and GU's

```
LIMIT 0,120,0,60
LOCATE 50,150,25,75
SCALE 0,10,0,10
```

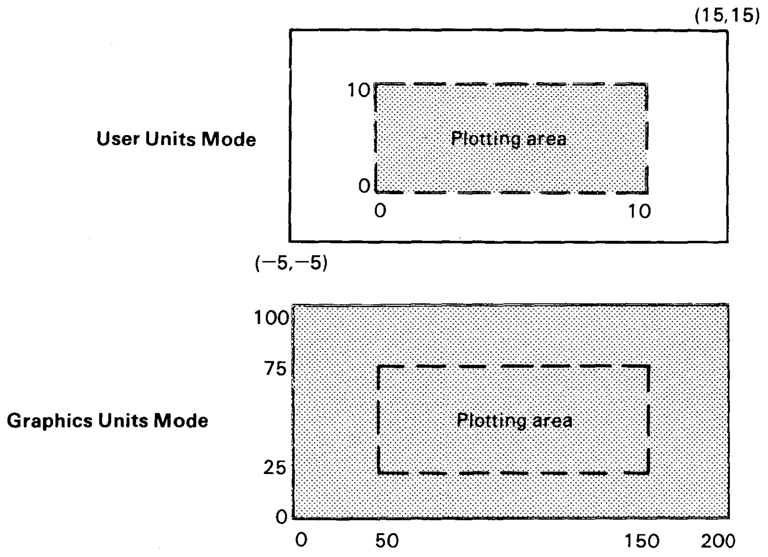


Figure 15-20b. LOCATE, UU's, and GU's

```
LIMIT 0,120,0,60
LOCATE -50,250,-50,150
SCALE 0,10,0,10
```

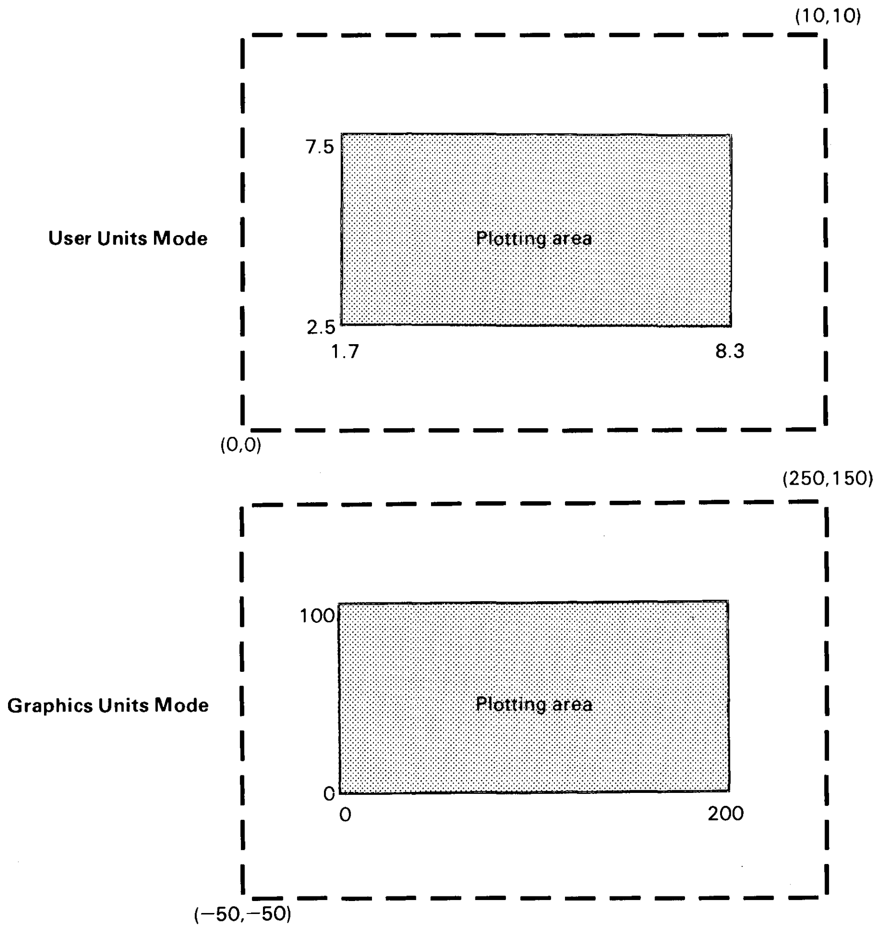


Figure 15-20c. LOCATE, UU's, and GU's

The following program sequentially frames the default graphics limits, the graphics limits specified by a LIMIT statement, and the LOCATE-specified plotting boundaries.

```
100 ! *** Locate ***
110 PLOTTER IS 1
120 GRAPHICS
130 FRAME !                  Frames the default graphics limits.
140 LIMIT 10,150,10,50
150 LINE TYPE 3,2 !          Specifies a dotted line type.
160 FRAME !                  Frames the specified graphics limits.
170 FOR I=1 TO 49 STEP 2
180   LOCATE 50-I,50+I,50-I,50+I ! Plotting boundaries in
190 !                              increments of 2 GUs.
200  LINE TYPE 1 !          Specifies a solid line type.
210  FRAME
220 NEXT I
230 END
```

Execution of the previous program results in the following display:

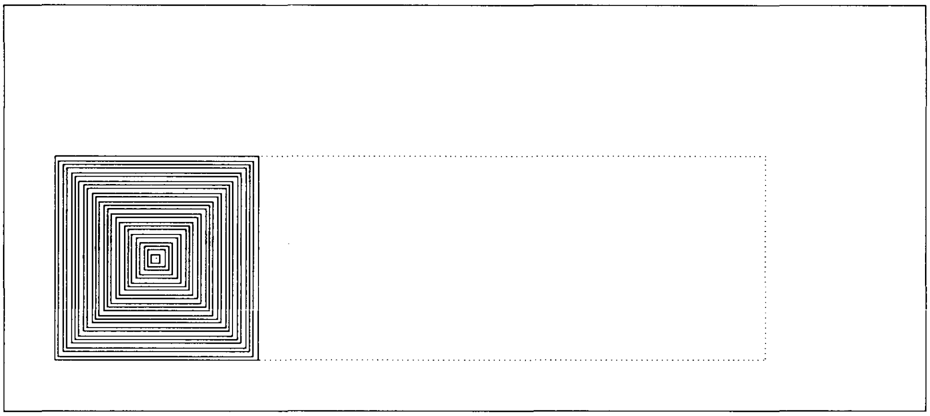


Figure 15-21. Limits and Plotting Bounds

CLIP Boundaries

The `CLIP` statement specifies the plotting boundaries according to the currents units: GU's, or UU's. Previously set plotting boundaries are replaced by the `CLIP`-defined boundaries. Plotting boundaries set by `LOCATE` or `CLIP` affect lines plotted in user units mode, but have no effect on lines plotted in graphics units mode or labels. The syntax for the `CLIP` statement is as follows:

```
CLIP Xmin,Xmax,Ymin,Ymax
```

The first two parameters specify the left and right plotting boundaries, respectively, and the second two parameters specify the lower and upper plotting boundaries, respectively. The parameters can be numeric constants, variables, or expressions.

The `CLIP` parameters are interpreted according to the current units, in contrast to the `LOCATE` statement which always uses GU's. The plotting area defined by the `CLIP` statement cannot be scaled by any of the three scaling statements: `SCALE`, `SHOW`, and `MSCALE`. If a scaling statement is executed after the `CLIP` statement, the user units scale is mapped onto the current plotting area as defined by the graphics limits or, if specified, the `LOCATE` plotting boundaries.

The graphics units scale is mapped onto the plotting area defined by the graphics limits. For example:

```
100 ! *** Clip ***
110 PLOTTER IS 1
120 GCLEAR
130 FRAME !                Frames the default plotting area.
140 LOCATE 10,90,10,70 !   Locates the plotting boundaries.
150 FRAME !                Frames the LOCATE plotting area.
160 CLIP 50,120,50,90 !    Specifies new CLIP plotting boundaries.
170 FRAME !                Frames the CLIP plotting area.
180 SCALE 0,5,0,5 !        Scales the LOCATE plotting area.
190 GRID 1,1 !            Draws a grid within the CLIP-defined
200 !                     plotting area according to the scale
210 !                     mapped onto the LOCATE-defined area.
220 LOCATE 10,90,10,70 ! Returns plotting boundaries to original
230 !                     LOCATE-defined position.
240 LINE TYPE 3 !         Specifies a dotted line type.
250 GRID 1,1 !            Draws a grid on the LOCATE plotting area.
260 END
```

Execution of the previous program results in the following display:

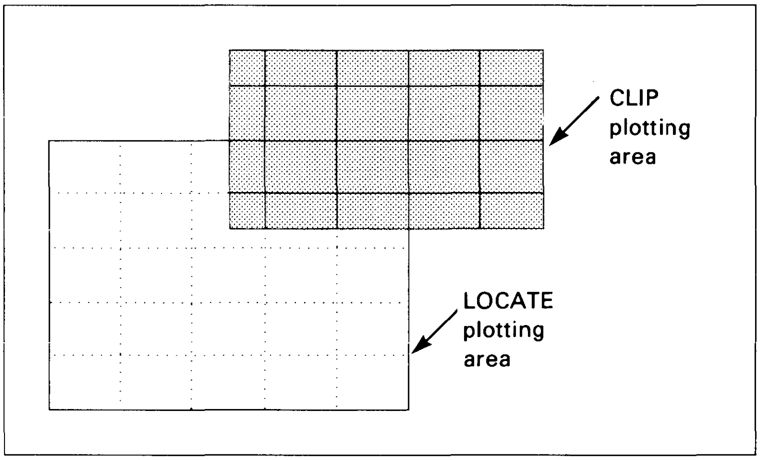


Figure 15-22. CLIP Boundaries vs. LOCATE Boundaries

The following program uses the CLIP statement to specify plotting boundaries and demonstrates plotting in graphics units mode and user units mode.

```
100 ! *** Clip-Plot ***
110 PLOTTER IS 1
120 GRAPHICS
130 GCLEAR @ FRAME
140 LIMIT 10,110,5,70 !                    Set new graphics limits.
150 LINE TYPE 6 @ FRAME
160 CLIP 10,RATIO*100-10,25,75 !           Plotting boundaries in
170 !                                      GUs - the current user
180 !                                      units scale.
190 LINE TYPE 3 @ FRAME
200 LINE TYPE 1
210 MOVE 0,100
220 FOR X=5 TO RATIO*100 STEP 5
230  IF X<50*RATIO THEN SETGU ELSE SETUU ! Sets graphics units mode
240  !                                    for left half of plot,
250  !                                    user units mode for right
260  !                                    half of plot. Plotting
270  !                                    scale is GUs for both
280  !                                    modes.
290  IF INT((-1)^(X/5))=1 THEN Y=100 ELSE Y=0
300  DRAW X,Y
310 NEXT X
320 ! *** Labeling is not restricted by the plotting boundaries ***
330  !
340 MOVE 3,10 @ LABEL "Graphics Units Mode"
350 MOVE 85,10 @ LABEL "User Units Mode"
360 END
```

Execution of the previous program results in the following display:

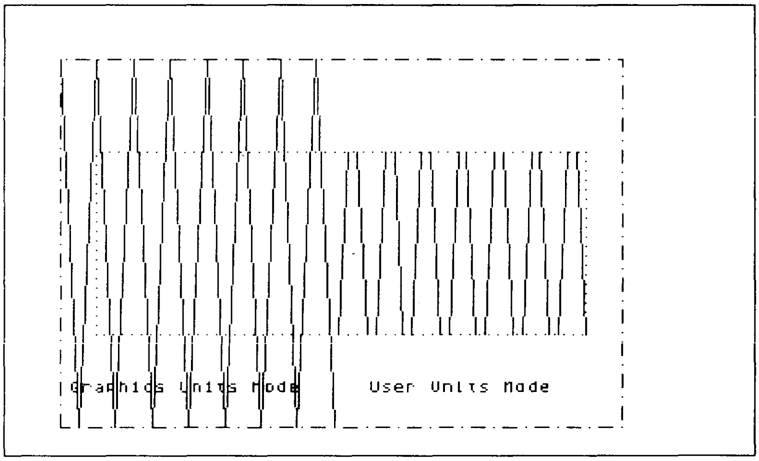


Figure 15-23. CLIP, GU's, and UU's

Unclipping Plotting Boundaries

The `UNCLIP` statement sets the plotting boundaries equal to the graphics limits, establishing the area bound by the graphics limits as the current plotting area. The syntax of this statement is as follows:

```
UNCLIP
```

`UNCLIP` doesn't disturb the current units; the system remains in the current scaling units mode. The `SETGU` statement also establishes the area within the graphics limits as the current plotting area, but without resetting the plotting boundaries. `SETGU` sets the system to graphics units mode.

The UNCLIP statement is used in the following program to reset the LOCATE-specified plotting boundaries to the graphics limits. The user units scale is preserved.

```
100 ! *** Unclip ***
110 PLOTTER IS 1
120 GCLEAR
130 !
140 LIMIT 0,115,0,75 @ FRAME ! Specifies and frames graphics limits (mm).
150 LOCATE 40,120,20,80 @ FRAME ! Locates and frames the plotting
160 !                              boundaries (GU's).
170 SCALE 0,10,0,10 !         Scales the LOCATE plotting area (UU's).
180 GRID 1,1 !                Draws a grid on the LOCATE area.
190 !
200 CSIZE 9,0.9 !             Specifies character size.
210 MOVE 2.2,-1.9 !           Moves the pen outside the plotting
220 !                         boundaries.
230 !
240 DISP "Program paused.  Type CONT to continue." @ PAUSE
250 !
260 LABEL "UNCLIP" !         Labels the character string "UNCLIP".
270 !
280 UNCLIP !                 Sets the plotting boundaries (LOCATE)
290 !                        back to the graphics limits (LIMIT).
300 GRID 1,1 !               Draws a grid on the plotting area
310 !                        bounded by the graphics limits; UUs
320 !                        are unchanged.
330 END
```

Execution of the previous program results in the following display:

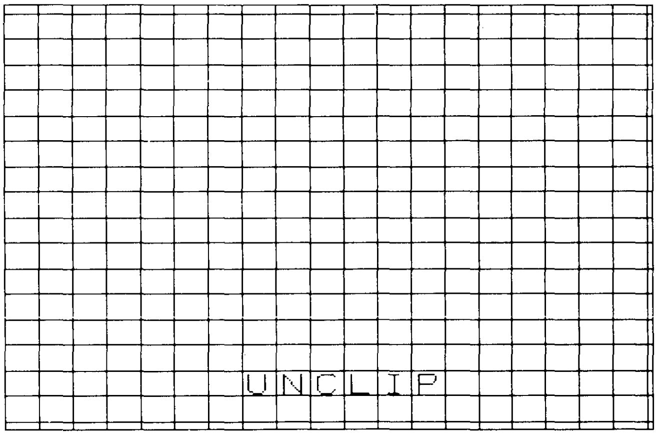


Figure 15-24. UNCLIP Resets Default Plotting Bounds

The following table summarizes the statements and conditions which affect the position and scale of the plotting area.

Table 15-1. Graphics Boundaries and Scaling

Condition or Statement	Parameter Units	Effect on Mode GU's vs. UU's	Effect on Scaling Units	Effect on Graphics Limits	Effect on Plotting Boundaries
Power-on	–	Set to UU's mode	UU's=GU's	Set to default graphics limits of the graphics display.	Set to default graphics limits of the graphics display.
PLOTTER IS	–	Set to UU's mode	UU's=GU's	Read from device	Set to graphics limits.
LIMIT	millimetres	Set to UU's mode	UU's=GU's	Set to specified limits.	Set to graphics limits specified by LIMIT.
LOCATE	GU's	No effect	No effect	No effect	Set to boundries specified by LOCATE.
CLIP	Current units	No effect	No effect	No effect	Set to boundries specified by CLIP
UNCLIP	–	No effect	No effect	No effect	Resets to current graphics limits
SCALE	UU's	Set to UU's mode	UU's specified by SCALE.	No effect	No effect
SHOW	UU'S	Set to UU's mode	UU's specifed by SHOW (in equal x& y units).	No effect	No effect
MSCALE	millimetres	Set to UU's mode	UU's specified by MSCALE in millimetres.	No effect	No effect
SETGU	–	Set to GU's mode. Plotting area is defined by graphics limits.	GU's	No effect	Temporarily set to graphics limits
SETUU	–	Set to UU's mode. Plotting area is defined by plotting boundries.	Current UU's as specified by the above statements and conditions.	No effect	Restores plotting boundaries

Reflecting Images

The normal sequence of parameters in the LIMIT statement puts the origin of your graph in the lower-left corner of the graphics output. By changing the order of parameters, you can produce a reflected image of the plot (except labels) without any additional changes in the program. Three kinds of reflected images are possible:

Exchanging the x_min with the x_max parameter reflects the image across the y axis.

```
   LIMIT x_max,x_min,y_min,y_max
```

Exchanging the y_min with the y_max parameter reflects the image across the x axis.

```
   LIMIT x_min,x_max,y_max,y_min
```

Exchanging the x_min with the x_max parameter, and the y_min with the y_max parameter reflects the image across origin.

```
   LIMIT x_max,x_min,y_min,y_max
```

The SCALE, SHOW, and LOCATE statements can also be used to reflect plots by exchanging parameters similarly.

NOTE

Note that BPLOT images are not reflected. In some cases, labels are not reflected; you can explicitly reflect labels with the CSIZE statement.

Digitizing Graphics Limits and Plotting Bounds

When executed without parameters, the `LIMIT`, `LOCATE`, and `CLIP` statements allow you to manually move the graphics limits or plotting boundaries on the plotting device. Executing these statements without parameters suspends program execution.

- With the `LIMIT` statement, the system waits to receive a message from the graphics input device containing the location of the lower-left and upper-right **graphics limits**.
- With `LOCATE` and `CLIP`, the system waits to receive the location of the lower-left and upper-right `CLIP` or `LOCATE` **plotting boundaries**.

The procedure for digitizing the graphics limits or the plotting boundaries is as follows:

1. Execute `LIMIT`, `LOCATE`, or `CLIP`, which suspends program execution.
2. Move the input locator (or pen) to the desired lower-left limit or boundary and press the "digitize button." The locator's coordinates are sent to the system, where they are interpreted as the lower-left limit or plotting boundary. The system beeps (if hardware is installed) when the button is pressed to signify that the digitized information has been received.
3. Move the input locator (or pen) to the desired upper-right limit or boundary and press the digitize button again. The locator's coordinates are sent to the system and interpreted as the upper-right graphics limit or plotting boundary. The computer beeps once again, if possible, after pressing the digitize button to signify that the digitized information has been received.
4. The digitized graphics limits or plotting boundaries are now active, and program execution continues.

Normally you would want to enter the lower-left limit or boundary first and the upper-right limit or boundary second. However, you can also digitize the graphics limits or plotting boundaries in different orientations to get a reflected image of your plot. For example, if you enter the upper-right limit first and the lower-left limit second, your plot will appear as if it was reflected through the origin. The procedure is analogous to exchanging parameters in the `LIMIT` or `LOCATE` statement. The sequence (first or second) and location (lower-left, upper-right, upper-left, or lower-right) of the digitized graphics limit or plotting boundary corner determines the type of reflection. The three types of reflections are summarized in the table below. (Note that digitized `CLIP` boundaries cannot be used to reflect plots.)

LIMIT and LOCATE Reflected Plots

	Unreflected plot	Reflection across origin	Reflection across x-axis	Reflection across y-axis
Location of first digitized limit or boundary	lower-left corner	upper-right corner	upper-left corner	lower-right corner
Location of second digitized limit or boundary	upper-right corner	lower-left corner	lower-right corner	upper-left corner

The following program digitizes the graphics limits, frames the plotting area, and then draws an arrow; the arrow points from the first digitized corner to the second digitized corner of the graphics limits. Experiment with your plotter by digitizing different graphics limits, and note how the shape and orientation of the figure changes.

```
100 ! ASSIGN 5 to "hpgl,/dev/hp7475A"
110 PLOTTER IS 1 !     Specifies the plotting device.
120 !
130 Loop: !           Repeat digitizing limits.
140 CLEAR
150 DISP "Digitize graphics limits:"
160 DISP "  1. Move input locator to 1st point; press ENTER."
170 DISP "  2. Move input locator to 2nd point; press ENTER."
180 !
190 LIMIT !           Computer waits while you digitize the
200 !                 graphics limits from the plotter.
210 CLEAR
220 DISP "Plotting."
230 !
240 FRAME !           Frames the digitized plotting area.
250 FOR I=1 TO 8 !    Plot the arrow.
260   READ X,Y
270   IF I=1 THEN MOVE X,Y ELSE DRAW X,Y
```

```
280 NEXT I
290 !
300 FOR I=1 TO 5 !      Plot box in arrow.
310   READ X,Y
320   IF I=1 THEN MOVE X,Y ELSE DRAW X,Y
330 NEXT I
340 !
350 RESTORE !           Restore DATA pointer.
360 GOTO Loop
370 !
380 DATA 20,10,10,20,20,30,10,40,60,60,40,10,30,20,20,10
390 DATA 37,17,41,17,41,21,37,21,37,17
400 !
410 END
```

The example output below shows the image of two arrows, each of which is the reflection (through the origin) of the other. It was produced by first digitizing the lower-left and upper-right corners, followed by digitizing the upper-right and lower-left corners—the same points, but in reverse order.

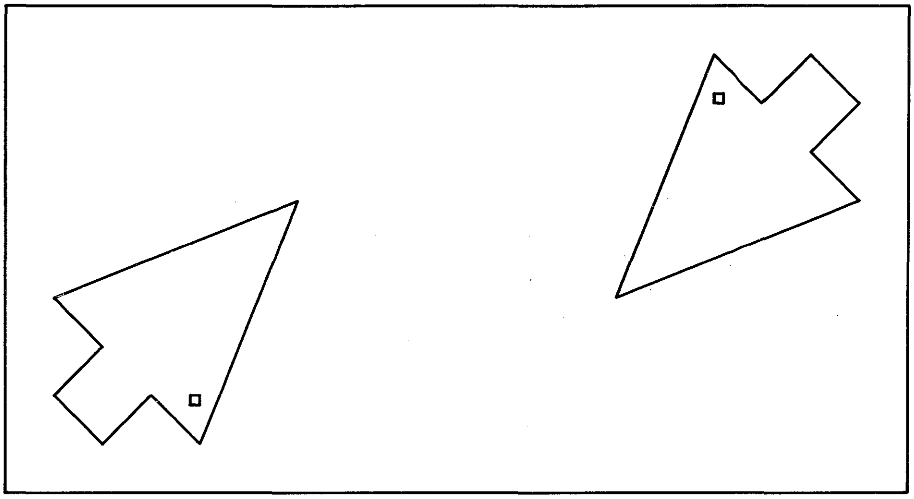


Figure 15-25. Reflecting Plots

Graphics Input

Technical BASIC has the ability to accept inputs from graphics devices. The "Example Graphics Programs" section at the beginning of the chapter showed a simple example of graphics input. This section further describes these operations.

Input Capabilities

The following capabilities are supported:

- Digitizing individual points using an input or output (plotting[1]) device, by first moving the input locator and then pressing the "digitize button." (If the graphics raster is on when `DIGITIZE` is executed, movements of the input locator will be "tracked" by a cursor on the output device.)
- Determining the input locator's current coordinates (without waiting for the "digitize button" to be pressed).

What Is Digitizing?

Digitizing is essentially the inverse of plotting.

- During plotting operations, the computer sends x,y coordinate values (and pen up/down instructions) to the plotting device, directing the pen to the specified location on the plotting area.
- During digitizing operations, you (optionally) move the graphics input locator to the desired point and then press a button to tell the system to determine the coordinates of the point (i.e., digitize it).

The digitizing process enables you to convert graphics information into digital information. For example, you could trace the outline of a drawing or photograph, digitizing points along the way.

Digitizing the Input Locator's Position

Digitizing the input locator's position is an operation which involves both the plotter and the computer. Here is an example statement:

```
DIGITIZE Xpos,Ypos
```

[1] See the "Using Plotters" section for specific details of digitizing using plotters.

The system first asks the input device for the locator's coordinates. When the "digitize button" is pressed, the device then sends the coordinates to the computer. The BASIC system then stores the information in the numeric variables specified. The variables are assigned values *according to the current scaling units.*

There is also another statement which can be used to digitize the input locator's location: `CURSOR`. Here is an example of using the `CURSOR` statement:

```
CURSOR Xpos,Ypos
```

where `Xpos` and `Ypos` are the variables that receive the coordinates of the input locator.

The syntax is the same as for the `DIGITIZE` statement, but the statements use different methods for entering the digitized information into computer memory:

- **The `DIGITIZE` statement suspends program execution** while you position the input locator to the desired location and waits until the "digitize button" is pressed. The locator's coordinates are read into computer memory only *after* the "digitize button" is pressed. (Tracking is enabled if the graphics raster is on when `DIGITIZE` is executed.)
- **The `CURSOR` statement does not suspend program execution.** The input locator's coordinates are read and placed in the specified variables immediately—without suspending program execution until the digitize button is pressed.

Therefore, the locator must be positioned at the desired position *before* executing the `CURSOR` statement. The `DIGITIZE` statement allows you to position the locator and enter the digitized information *after* `DIGITIZE` is executed.

NOTE

The optional *pen status* parameter available with `DIGITIZE` and `CURSOR` is meaningless when the graphics input device is separate from the graphics output device. It is only useful when the input and output devices are a single device; for instance, it can be useful when using many pen plotters, because the pen is both output device and input locator. Note, however, that this parameter is set by the last *plotting* operation, not by the last *input* operation. See the subsequent section called "Using Plotters" for additional details.

Pens and Background

Traditionally, drawing requires pen and paper. With raster graphics, the paper is replaced by the graphics raster, and the pen is replaced by software that turns raster pixels on and off.

The preceding examples did not need to specify a pen number, since the default is `PEN 1`, which draws a white line (on a black background).

Monochromatic Pens

On monochromatic graphics rasters, the `PEN` statement lets you choose between three different pens:

Table 15-2. Monochromatic Pens

Pen Number	Effect
`PEN 1`	white pen—turns pixels on
`PEN 0`	black pen—turns pixels off
`PEN -1`	complementing pen—white pixels are changed to black, and black pixels are changed to white (providing the display supports *block read/write* operations; see `ASSIGN` in the *BASIC Reference Manual* for a list of displays with this capability).

Color Pens

On color graphics rasters, the PEN statement is also the means of selecting any of the available color pens. For instance, here is the *default* set of colors for the HP 98700 Color Display Controller.

Table 15-3. Default Color Pens

Pen Number	Default Color
PEN 7	Magenta
PEN 6	Blue
PEN 5	Cyan
PEN 4	Green
PEN 3	Yellow
PEN 2	Red
PEN 1	White
PEN 0	Black
Negative pens[1]	Complementing pens

The background is normally black, and all pens write in "dominant" mode—that is, they overwrite any color (or black) that is currently on the screen[1].

Other displays may have a different set of color pens available[2].

[1] "Complementing" pens are available by using the negative of the pen number. For instance, on the 98700 display, dominant red is **PEN 2** and complementing red is **PEN -2**. The resultant operations for the complementing pens are as follows: for each pixel drawn on the screen, take the bits of the screen pixel (the *destination*) and **exclusive or** them with the bits of the pen color (the *source*); the resultant value is placed into the screen pixel (*destination*).

[2] With some color displays, you can also re-define the default *color map*. See "Section 3: Starbase Color Graphics" of the *HP-UX Concepts and Tutorials, Vol. 6: Graphics* manual for details. It is possible to either do this while in the HP-UX system and then enter BASIC, or to do it in a binary program that the BASIC system calls (see one of the "Binaries" chapters for examples of calling a routine written in another language).

Moving the Pen and Drawing

Several statements control the movement of the pen on the drawing surface.

`MOVE X,Y`	moves the pen to the coordinate `X,Y` (without drawing).
`DRAW X,Y`	draws a line from the current pen position to the coordinate `X,Y`.
`PLOT X,Y,PenCtrl`	moves or draws to point `X,Y` as directed by the value of `PenCtrl`.
`IMOVE X,Y`	moves to point `X,Y` using the *current* pen position as the origin.
`IDRAW X,Y`	draws to point `X,Y` using the *current* pen position as the origin.
`PLOT X,Y,PenCtrl`	moves or draws to point `X,Y` as directed by the value of `PenCtrl`.
`IPLOT X,Y,PenCtrl`	moves or draws to point `X,Y`, as directed by the value of `PenCtrl`, using the *current* pen position as the origin.
`RPLOT X,Y,PenCtrl`	moves or draws to point `X,Y`, as directed by the value of `PenCtrl`, using the last pen location (specified by another plotting statement) as the origin.

Moving and Drawing

Try the following example.

```
100 PLOTTER IS 1
110 GCLEAR
120 FRAME
130 DRAW 60,50
140 LABEL " X=60, Y=50"
```

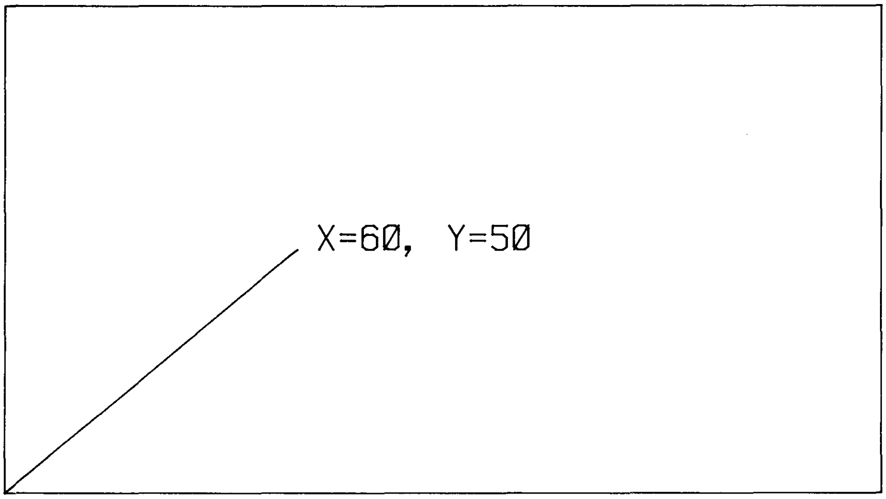


Figure 15-26. Drawing and Labeling

A white line is drawn from the lower left corner to the middle of the screen. Why from the lower left corner? Because executing the statement `PLOTTER IS 1` returns the pen to location 0,0. This is currently the lower left corner of the display.

Execute the following program to see these statements' effects. (If your display does not support *block read/write* operations[1], then this example will not work as described below.)

```
100 PLOTTER IS 1
110 !
120 PEN 1 !        White pen.
130 GCLEAR !       Clear graphics raster.
140 MOVE 0,50 !    Move to left center.
150 DRAW 100,50 ! Draw solid white line.
160 WAIT 3000 !    Wait 3 seconds.
170 !
180 MOVE 50,50 !   Move to center of line.
190 PEN 0 !        Change to black pen.
200 DRAW 0,50 !    Draw over left half of line.
210 WAIT 3000 !    Wait another 3 seconds.
220 !
230 PEN -1 !       Complementing pen.
240 DRAW 100,50 ! Draw over entire line.
250 !
260 END
```

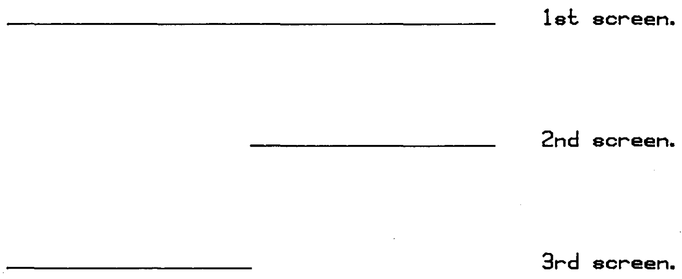


Figure 15-27. Using the Complementing Pen

A white line is first drawn across the screen. Then the pen is moved to the midpoint of the line, and a black pen draws over the left half of the line. Finally, the entire line is complemented.

[1] See **ASSIGN** in the *BASIC Reference Manual* for a list of devices with block read/write capability.

Pen Control with PLOT

Pen action is automatically defined for the MOVE and DRAW statements. The pen is always raised *before* a MOVE and lowered *before* a DRAW. The PLOT statement has an optional pen control parameter that determines the pen's action according to the following table.

Table 15-4. PLOT Pen-Control Parameters

Control Value	Result
Positive Odd	Move pen, then lower it
Positive Even	Move pen, then raise it
Negative Odd	Lower pen, then move it
Negative Even	Raise pen, then move it

Note that when a positive parameter is used, the pen's up/down status is *not* changed before moving it. For instance, if the pen is currently lowered and a postive pen control parameter is used in a PLOT, then the pen remains down *throughout* the entire operation; it is *not* raised before the move and then lowered after the move.

The following example shows controlling the up/down motion of the pen by using the optional pen control parameter.

```
100 PLOTTER IS 1
110 !
120 PEN 1 !         White line.
130 GCLEAR !        Clear graphics raster.
140 !
150 PENUP !         Make sure pen is rasied.
160 PLOT 20,20,1 !  Move, then lower.
170 PLOT 40,20 !    Draw (since pen lowered).
180 PLOT 40,40,0 !  Draw, then raise.
190 PLOT 20,20,1 !  Move, then lower.
200 PLOT 20,40 !    Draw again.
210 PENUP !         Raise pen.
220 !
230 END
```

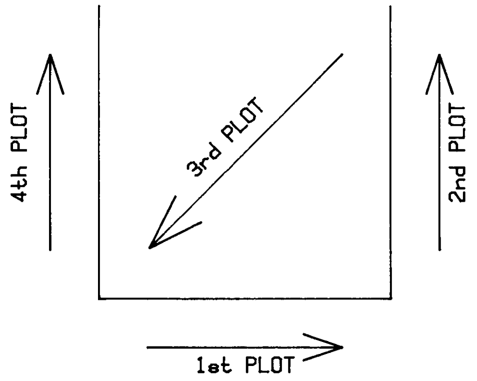


Figure 15-28. Using Pen Control Parameters with PLOT

Relative Plotting

A second method of moving and drawing involves using a new origin, and specifying pen movements relative to this origin. The relative plot (RPLOT) statement uses the current pen position as a new origin to define a second coordinate system. This new origin is located wherever the last plotting statement (*other than* RPLOT) left the pen. Since RPLOT uses a movable origin, it is useful when drawing a figure that needs to be repeated at different locations on the display.

Here is an example usage of the statement:

```
RPLOT X,Y,PenControl
```

X and Y specify relative displacements from the *current* RPLOT origin.

This program draws a triangle at three different locations.

```
100 PLOTTER IS 1
110 GCLEAR @ FRAME
120 !
130 MOVE 50,50 !   Sets the relative origin.
140 GOSUB Triangle
150 !
160 MOVE 10,10 !   Move the relative origin.
170 GOSUB Triangle
180 !
190 MOVE 80,80 !   Move it again.
200 GOSUB Triangle
210 !
220 STOP
230 !
240 Triangle: ! Draw using relative coordinates.
250   RPLOT 20,10,-1
260   RPLOT 20,0
270   RPLOT 0,0,2 ! Pen up after draw.
280 RETURN
```

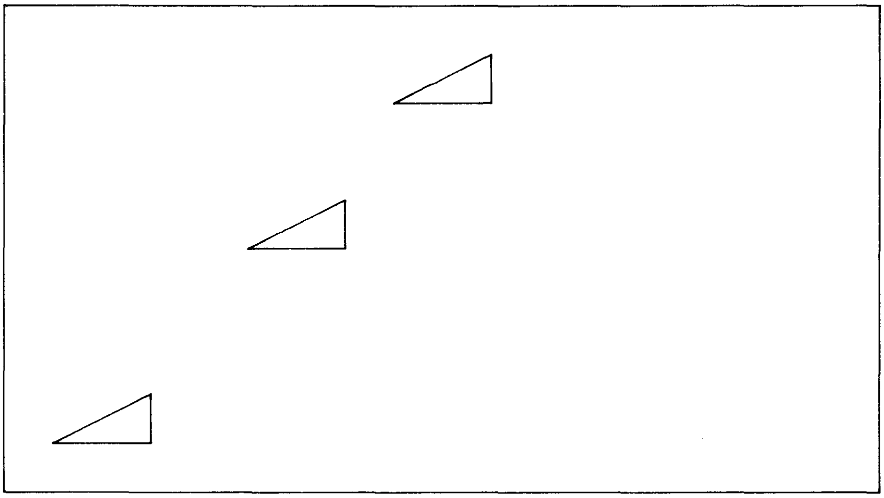


Figure 15-29. Relative Plotting (RPLOT)

Note that the command `RPLOT 0,0` returns the pen to the "local" origin (e.g. 50,50) not the "absolute" origin (0,0).

Incremental Pen Positioning

It also useful in some situations to have statements that define the pen's *current* location as a new origin. Plotting coordinates are then specified relative to this new origin, which is moved *every* time the pen is moved.

```
IDRAW X,Y

IMOVE X,Y

IPLOT X,Y,P
```

This type of plotting is similar to RPLOT, except that *every* pen movement defines a new origin—including those produced by IDRAW, IMOVE, and IPLOT.

Execute the following program and watch the results.

```
10 PLOTTER IS 1
20 !
30 MOVE 40,40
40 IDRAW 30,0 !  Draw right.
50 IDRAW 0,30 !  Draw up.
60 IMOVE -30,0 ! Move left.
70 IDRAW 0,-30 ! Draw down.
```

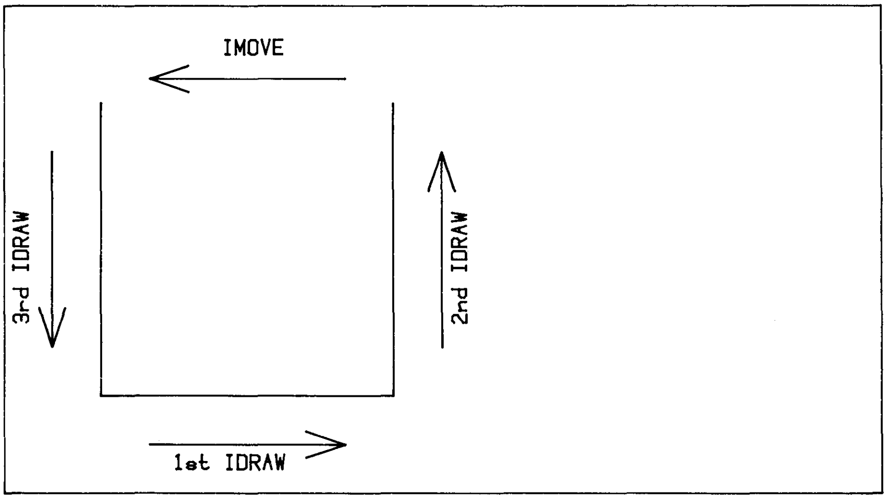


Figure 15-30. Incremental Plotting (IDRAW)

With *each* incremental movement of the pen, a *new* origin is created for the subsequent incremental-plotting (IPLOT) statement.

Rotating Incrementally Plotted Lines

Lines generated by incremental plotting (IDRAW, IMOVE, and IPLOT) can be rotated by the PDIR (plot direction) statement. The current angle mode determines how the angle parameter is interpreted; in the following example, the DEG statement specifies that the angle parameter of PDIR is to be interpreted in degrees.

```
100 PLOTTER IS 1
110 GCLEAR @ FRAME
120 !
130 DEG ! Use angular mode of degrees.
140 FOR Angle=0 TO 90 STEP 10
150   PDIR Angle
160   MOVE 0,0
170   IDRAW 100,0 ! Rotated Angle degrees.
180 NEXT Angle
```

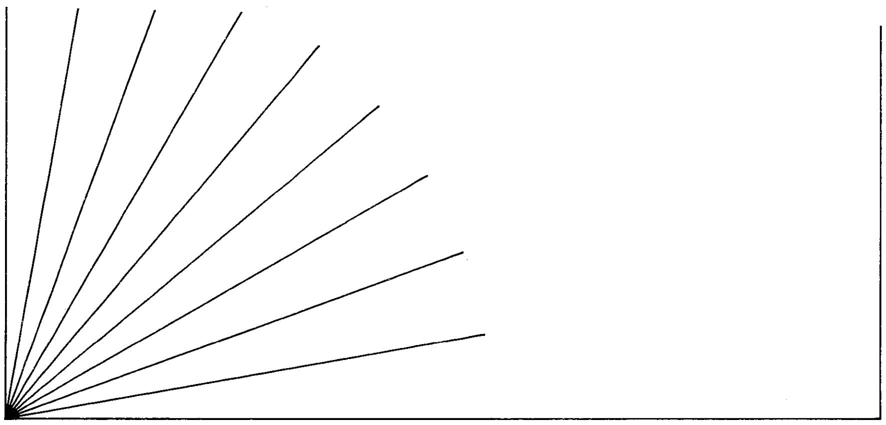


Figure 15-31. Rotating Plots

PDIR 0 will return to the system to normal (no rotation).

Line Types

There are eight different types of lines available with the `LINE TYPE` statement. Examples are solid, dashed, dotted, and alternating dashes and dots. The *HP-UX Technical BASIC Reference Manual* shows examples of each type.

Two parameters are allowed with the `LINE TYPE` statement:

- line type (default is 1)
- repeat length (default is 5).

Thus, the default is `LINE TYPE 1,5`. (The repeat length cannot be changed on some Series 200/300 and 500 devices.)

```
100 PLOTTER IS 1
110 !
120 FOR LineType=1 TO 8
130   LINE TYPE LineType
140   MOVE 10,90-10*LineType
150   DRAW 90,90-10*LineType
160 NEXT LineType
170 END
```

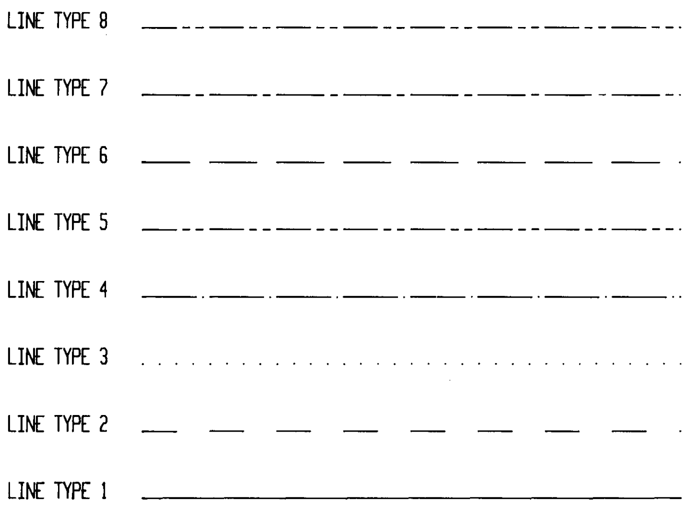


Figure 15-32. A Sample of Line Types

When the graphics raster is used as the plotting device, the repeat factor is system-dependent.

Labeling

Although images can convey a great deal of information, a few labels help explain what is being presented. The following program places a label on the graphics raster.

```
10 PLOTTER IS 1
20 GCLEAR @ FRAME
30 !
40 MOVE 50,50
50 CSIZE 12 ! Large characters.
60 LABEL "Sin X"
```

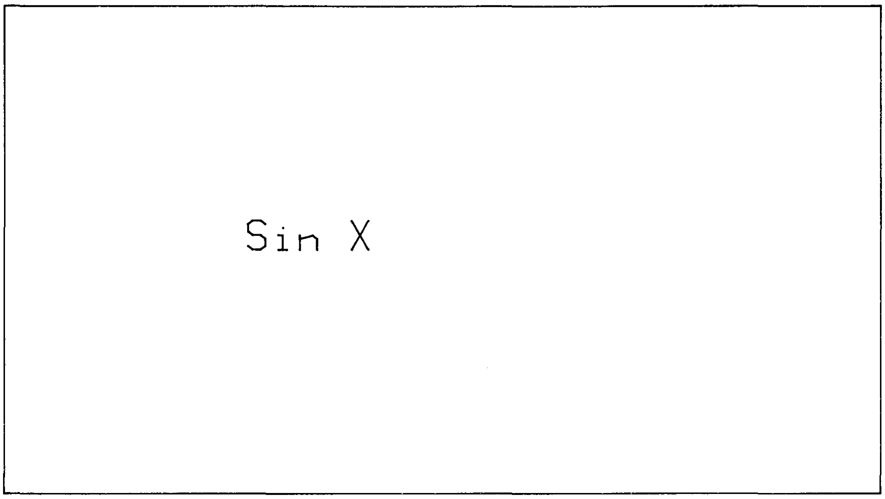


Figure 15-33. Simple Example of Labeling

Table 15-5. Statements that Affect the Printing of Labels

CSIZE	controls the character size, aspect ratio, and slant.
LDIR	(label direction) specifies the printing angle of the label.
LORG	(label origin) adjusts the location of the label.

Each of the following examples illustrates one of the above statements.

```
100 PLOTTER IS 1
110 GCLEAR @ FRAME
120 !
130 MOVE 15,40
140 CSIZE 10,4 !  Height 10; aspect ratio 4
150 LABEL "WIDE"
160 !
170 MOVE 15,15
180 CSIZE 20,0.2 ! Height 20; aspect ratio=0.2
190 LABEL "TALL"
```

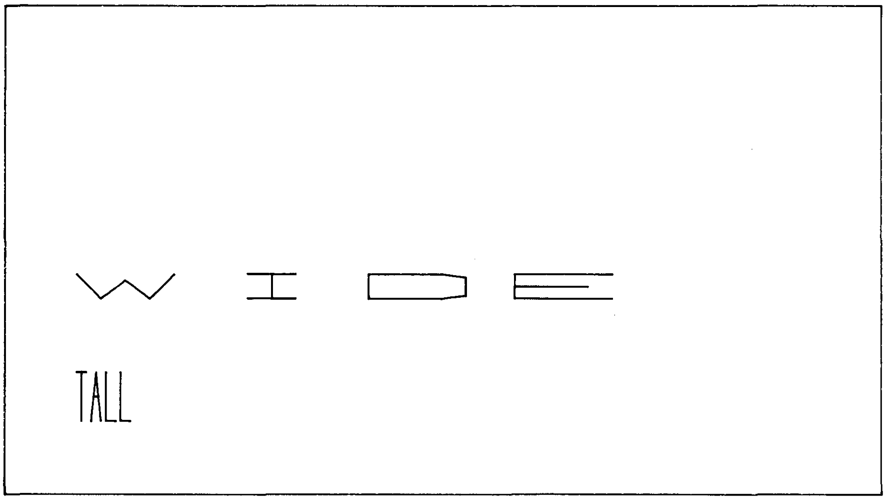


Figure 15-34. Changing the Size of Graphics Labels

Label direction is interpreted according to the current angular mode: degrees, radians, or grads.

```
100 PLOTTER IS 1 @ FRAME
110 !
120 DEG ! Set degrees angular mode.
130 !
140 MOVE 90,30
150 CSIZE 9
160 LDIR 90 ! Label direction is bottom to top.
170 LABEL "VERTICAL"
180 !
190 MOVE 80,30
200 LDIR 180 ! Label direction is right to left.
210 LABEL "UPSIDE DOWN"
220 END
```

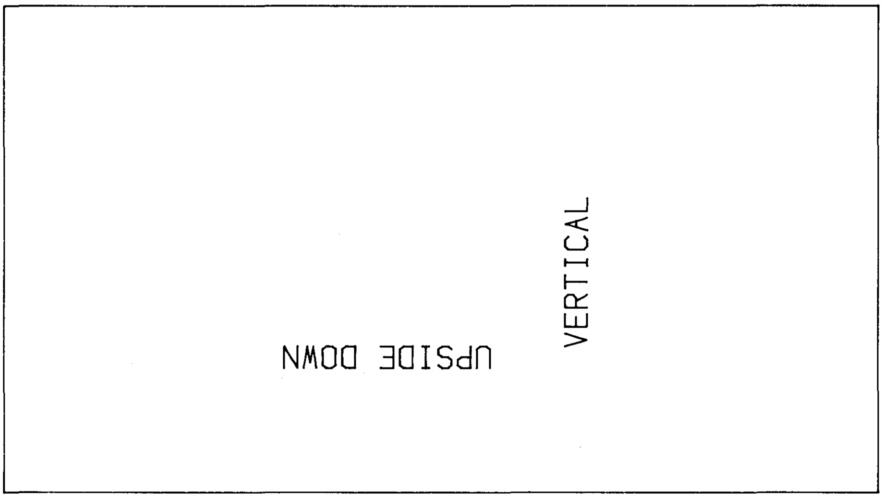


Figure 15-35. Changing Graphics Label Direction

There are nine possible label origins used for adjusting the location of the label. This program shows three. (See the *HP-UX Technical BASIC Language Reference* for a description of all nine.)

```
100 PLOTTER IS 1
110 GCLEAR @ FRAME
120 CSIZE 6
130 !
140 MOVE 60,70
150 LORG 1 !     1st "label origin" statement.
160 LABEL USING "K"; "RIGHT"
170 !
180 MOVE 60,50
190 LORG 5 ! 2nd "label origin" statement.
200 LABEL USING "K"; "CENTER"
210 !
220 MOVE 60,30
230 LORG 9 !  3rd "label origin" statement.
240 LABEL USING "K"; "LEFT"
250 END
```

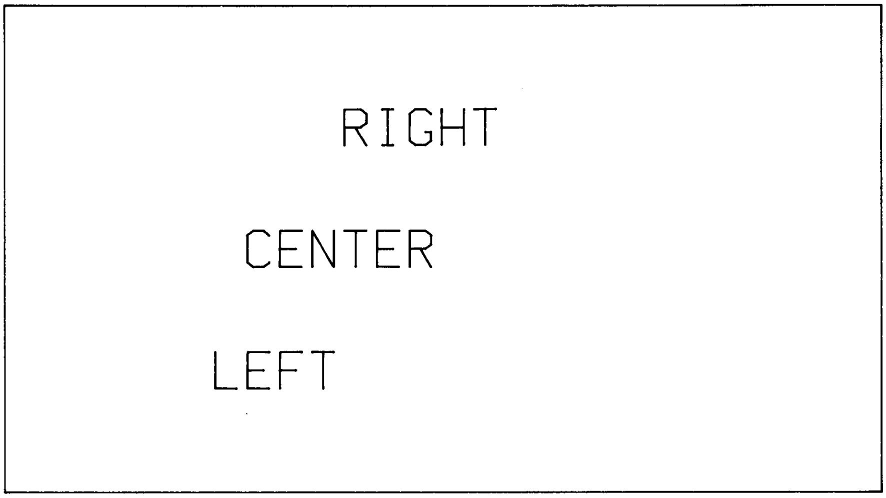


Figure 15-36. Changing the Graphics Label Origin

Note that the BASIC system uses default widths of 21 characters with some types of data in labels. The effects of this convention are not apparent when using `LORG` values 1 through 3, because these values specify that the field is to be left-justified (and trailing blanks do not show up). However, they may become a problem when using `LORG` values of 4 through 9, since these values specify that text is to be centered or right-justified in these 21-character fields. You can suppress these additional characters by using `LABEL USING "K";...`, since `LABEL USING` allows formatted labels to be plotted just as `PRINT USING` allows formatted text to be printed.

Plotting and Reading Pixels

This section describes a special plotting operations that are available on raster devices that support *block read/write* operations. These operations are also called "bit block transfers."

Plotting Pixels (BPLOT)

The BPLOT, or byte plot, statement performs a type of plotting operation in which the system accesses individual pixels on the graphics raster, changing them according to the parameters in the BPLOT statement.

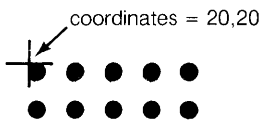
Here is the general form of the statement:

BPLOT *byte_plot_string*, *pixels_per_row*

Here are two examples:

```
WhitePen$=CHR$(1)
TenWhitePixels$=RPT$(WhitePen$,10)
FivePerRow=5
MOVE 20,20
BPLOT TenWhitePixels$,FivePerRow
```

Plots this:



```
RedPen$=CHR$(2)
TwelveRedPixls$=RPT$(RedPen$,12)
SixPerRow=6
MOVE 30,30
BPLOT TwelveRedPixls$,SixPerRow
```

Plots this:

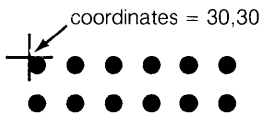


Figure 15-37. Plotting Pixels

Reading Pixels (BREAD)

The BREAD statement allows you to read pixels on the graphics raster. The raster is read dot by dot and placed into a character string.

```
DIM FourByFive$ [20]
MOVE 50,50
BREAD FourByFive,4
```

Reads this:

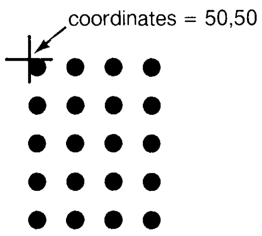


Figure 15-38. Reading Pixels

BREAD performs the opposite function as BPLOT; the two statements are often used cooperatively for creating and storing graphics raster images.

A Closer Look at Byte Plotting

This section elaborates on how BPLOT works.

How the String Parameter Is Used

With the BPLOT statement, each character in the string expression specifies the color of one pixel on the graphics raster.

Starting Location

BPLOT can begin at any pixel location. The starting location for BPLOT is determined in two ways:

- If the most recent pen movement was directed by any statement other than BPLOT, then the BPLOT begins at the current pen position (or the closest pixel to it).
- If the most recent pen movement was directed by a BPLOT statement, then the next BPLOT string begins one row below the left end of the last byte-plotted string.

The BPLOT statement doesn't affect the pen position for other plotting operations. On the other hand, all of the other plotting statements which move the pen *do* affect the beginning location of subsequent byte-plotted information. (Note that byte plots can't be reflected by changing graphics limits or scaling, as can other plotted images.)

Number of Pixels Per Row

The *pixels_per_row* parameter specifies the number of pixels per row; it can be a numeric constant, variable, or expression.

- If the *pixels_per_row* parameter is *positive*, the BPLOT statement performs an *exclusive or* with the existing dots on the display screen.
- If it is *negative*, the pixels are *overwritten* by the pen number specified in the corresponding string character. (This is also known as a "dominant" pen.)

When the specified number of pixels per row are plotted, BPLOT repositions the pen to the left end of the row just plotted, then moves down one row. BPLOT continues to plot pixels until the entire character string is used to plot pixels.

Additional Details

- The plotting area is set up using the same procedures as for plotting data, axes, and labels.
- You can start a BPLOT operation anywhere within the current graphics limits (default limits, or those specified by a LIMIT statement). The operation may even spill over into an area outside these limits; however, the pen cannot be positioned outside the current graphics limits prior to BPLOT, unless the system is set to graphics units (GU) mode.
- The PEN statement is ignored during BPLOT operations. BPLOT plots dots according to the BPLOT's string parameter (the first parameter).

A Closer Look at Byte Reading

You can read the current states of pixels on the raster by using the BREAD statement. Here is the general form of the statement:

BREAD *byte_read_string*, *pixels_per_row*

The BREAD (byte read) statement performs the converse of BPLOT: it reads pixels from the graphics display and stores them as characters in the string variable—one pixel per character.

How the String Parameter Is Used

With the BREAD statement, each character in the string expression receives the pen number of one pixel on the graphics raster.

Starting Location

The byte reading begins at the current pen position, reading rows of pixels from left to right. The pen number of each pixel is read, and placed into the string, beginning at the first character and sequentially moving through the string.

After reading one row of the specified number of pixels, the statement moves down one row of dots and begins reading that row. The `BREAD` statement continues to read bytes across and down — building the character string until the string variable has reached its allocated length. Recall that strings longer than 18 characters must be allocated memory through a `DIM` statement.

Number of Pixels Per Row

The *pixels_per_row* parameter can be a numeric constant, variable, or expression; negative values are interpreted as their absolute value.

`BREAD` does not effect the pen location for any plotting operation other than `BREAD` and `BPLOT`. An example of using `BREAD` is as follows:

```
BREAD String$,32
```

where *String$* is the string variable that is to receive the pixels to be read, and `32` is the number of pixels per row.

Additional Details

- The plotting area is set up using the same procedures as for plotting data, axes, and labels.
- You can start a `BREAD` operation anywhere within the current graphics limits (default limits, or those specified by a `LIMIT` statement). The operation may read pixels from an area outside these limits; however, the pen cannot be positioned outside the current graphics limits prior to `BREAD`, unless the system is set to graphics units (GU) mode.

Storing and Retrieving Raster Images

One handy feature of using the graphics raster as a plotting device is that the image can be stored in a file and later retrieved[1], or dumped to a compatible printer[2].

Storing Graphics in a File

There are two methods of performing this operation:

- Use GSTORE to store the entire raster in a BASIC/GRAF file.
- Use BREAD to read a portion of the raster, then write the information in a file (with OUTPUT or PRINT#).

Using GSTORE

The following statement stores the current graphics raster in the file named LORG_Raster.

```
GSTORE "LORG_Raster"
```

The statement creates a file of type BASIC/GRAF in the current working directory, and then stores the pixels in the file.

Retrieving Graphics from a File

Like storing the graphics raster, there are two ways to retrieve a stored raster:

- Use GLOAD to load a GSTORE'd raster in a BASIC/GRAF file.
- Read the information in a file (with ENTER or READ#). Use BPLOT to write the corresponding portion of the raster.

[1] GSTORE, GLOAD, and DUMP GRAPHICS are only possible with graphics raster devices which support *block read/write* operations. See ASSIGN in the *BASIC Reference Manual* for a list of devices with this capability.

[2] The printer must support the "HP Raster Interface Standard" for graphics dumps, in addition to the graphics raster display supporting block read/write operations. See your printer's manual, or an HP Configuration Guide, to determine whether your printer has this capability.

Using GLOAD

The image can be returned to the graphics raster from a file by this statement:

```
GLOAD "LORG_Raster"
```

The `BASIC/GRAF` file's contents are loaded back into the raster.

The general rule with `GSTORE` and `GLOAD` is to store and load from the **same size** display screen or window. For instance, if you `GSTORE` a 512×390 graphics screen, then you should `GLOAD` it back into the same size screen. Attempting to `GSTORE` an image on a particular size of screen and then `GLOAD` it back into a *different* size of screen or window will not generally work.

Dumping Graphics to Printers

You cannot directly send graphics commands to a printer like you can to a plotter. However, printers that support the "HP Raster Interface Standard[1]", such as the Thinkjet and Laserjet printers, are capable of reproducing on paper the image on the graphics raster. Usually there is a one-for-one correspondence between a pixel on the screen and a dot on the printed paper.

Execute the `DUMP GRAPHICS` statement to dump the graphics raster to a printer.

```
DUMP GRAPHICS [Return]
```

NOTE

Since you will be using a "raw" device (not a "spooled" device), you will need to make sure nobody else is using the printer during the dump.

[1] `DUMP GRAPHICS` is only possible with graphics raster devices which support *block read/write* operations. See `ASSIGN` in the *BASIC Reference Manual* for a list of devices with this capability.

In addition, the printer must support the "HP Raster Interface Standard" for graphics dumps, in addition to the graphics raster display supporting block read/write operations. See your printer's manual, or an HP Configuration Guide, to determine whether your printer has this capability.

Using Plotters

Now that you have used most of the features of graphics raster devices, it is time to expand this knowledge to include pen plotters. Most of the operations described in preceding sections are applicable to using plotters. However, there are a few differences and additional capabilities with plotters. This section describes the additional considerations that you will need to make when using plotters.

Graphics Defaults Restored

When you change plotting devices (with `PLOTTER IS`), the system sets up certain default conditions on the device and in the BASIC graphics system itself. See `PLOTTER IS` in the *Technical BASIC Reference Manual* for details. (Note that the current pen may not be affected by executing `PLOTTER IS`.)

The following operations also set up default graphics conditions:

- Executing a `LIMIT` statement.
- Exiting and re-entering BASIC.

For a complete list of graphics default conditions, refer to the *HP-UX Technical BASIC Reference*.

Additional Considerations

There are several special considerations when using an external plotter.

- For instance, `PENUP` lifts a plotter's pen from the surface of the paper. The logical pen used with raster graphics could care less where it sits, but real pens with real ink make a real mess unless lifted from the paper. Thus programs that use pen plotters should make a point of lifting the pen whenever it is not moving. After plotting, cap the pen (or execute `PEN 0` to put the pen away).

NOTE

`PENUP` may not be executed immediatly due to the buffering of commands by the HP-UX system.

- The aspect ratio of an external plotter is often different than the aspect ratio of a graphics raster. Character size is also affected by this difference.
- Lines drawn by the `LINE TYPE` statement will differ from those defined for the display. Check the plotter's manual for descriptions of line type parameters.

Digitizing Plotter Pen Locations

Since most plotters have both input and output capabilities[1], you can use the pen as an input device during digitizing operations. You can digitize any point on the plotting area and store its coordinates for later use. In order to better understand these operations, however, you may need a little background.

Physical and Logical Pens

The the ink pen on a pen plotter is considered to be a "physical pen," because it actually draws points, lines, and curves.

The BASIC system has a pen of its own, known as the "logical pen". The x and y coordinates and up/down status of the logical pen reside in memory and are determined by the most recently executed statement affecting logical pen location and status. For example, executing `MOVE 10,20` moves the logical pen to the coordinates 10,20 (the physical coordinates of which are determined the current scaling units).

On some devices, the physical pen location can be changed at the device. For example, you can move a plotter pen by using the front-panel pen-movement controls. The physical pen location can also be altered by executing a plotting statement (for example `PLOT`, `MOVE`, or `LABEL`). The physical pen is always located within the physical limits of the plotting device, but not necessarily within the current plotting area.

The location and status of the logical pen are unaffected by the pen movement controls on the plotting device. The logical pen can be located anywhere inside or outside the physical limits of the device.

1 The plotter must be specified as both input and output device; you cannot have *separate* input and output devices (such as a display and a mouse). See **ASSIGN** in the *Technical BASIC Reference Manual* for further information.

Although the physical and logical pens coincide with each other during most plotting operations, they are each recognized individually by the system. Listed below, are some instances where the logical and physical pens have different locations.

- Whenever the graphics default conditions are activated, the logical pen moves to the lower-left corner of the plotting area. However, the physical pen location is unaffected.
- When a plotting statement directs the pen to a point outside the current plotting area, the physical pen stops short of the intended point, at the current graphics limit or plotting boundary and is lifted (refer to the diagram below). In contrast, the logical pen location and status always coincide with the destination point and status specified by the plotting statement, regardless of whether or not the point lies within the current plotting area and whether or not it was actually plotted.
- Whenever the plotting device is changed, the physical and logical pens may have different locations depending on the initial physical pen position and the last executed plotting statement.
- Whenever the physical pen is moved using the pen movement controls at the external plotting device, the physical and logical pen locations differ.

The following diagram shows the location of the physical and logical pens during the sequence of plotting statements listed in the table below. The framed plotting area is scaled from 0 to 15 in the X direction and from 0 to 10 in the Y direction. The solid black line indicates a line drawn during physical pen movement. The dashed black line indicates physical pen movement without line drawing. Note that the computer plots successive points according to the logical pen location.

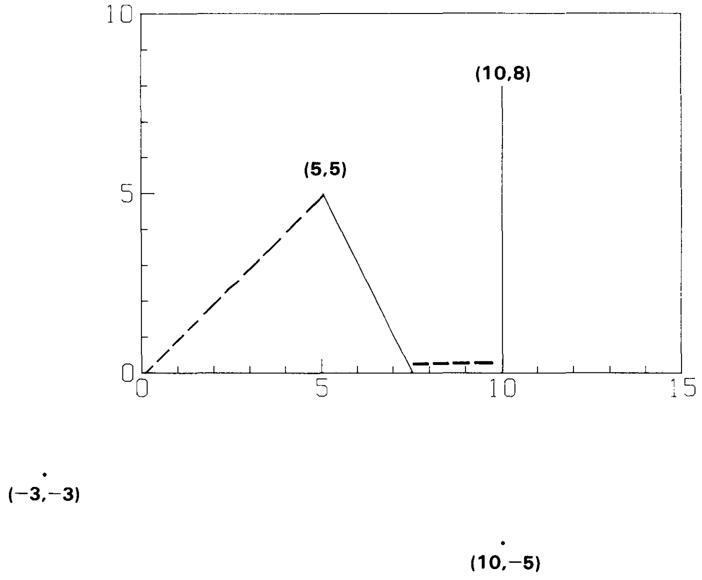


Figure 15-39. Physical and Logical Pen Locations

Table 15-6. Physical and Logical Pen Movement Operations

Execute:	Resulting physical pen location and status	Resulting logical pen location and status
`PLOT 10,8,1`	(10,8) down	(10,8) down
`DRAW 10,-5`	(10,0) up	(10,-5) down
`PLOT 5,5`	(5,5) down	(5,5) down
`MOVE -3,-3`	(0,0) up	(-3,-3) up

Digitizing the Physical Pen Location

Digitizing the plotter's physical pen is an operation which involves both the plotter and the computer. Here is an example statement that digitizes the PLOTTER IS device's *physical pen* location:

```
DIGITIZE Xpos,Ypos,PenStatus
```

The system first asks the plotter for the pen's coordinates and up/down status. When the [Enter] button is pressed, the plotter then sends the information (as numbers) to the computer. The BASIC system then stores the information in the three numeric variables specified. The first two variables identify the coordinate location of the physical pen; the third variable identifies the pen status. The variables are assigned values according to the current scaling units. The optional third variable parameter is assigned the pen status information. If the pen is up, 0 is assigned to the variable. If the pen is down, 1 is assigned to the variable. All three variables must be numeric variables.

NOTE

The PenStatus parameter does not reflect the current pen's up/down status. Instead, it is set by the last drawing operation that affected pen status. For instance, if the last drawing operation was a MOVE, then the PenStatus parameter would indicate that the pen is "up". If the pen was put down since the MOVE (by a manual operation at the plotter), then the PenStatus parameter will not reflect the pen's up/down status correctly.

There is also another statement which can be used to digitize the physical pen's location: CURSOR. The syntax is the same as for the DIGITIZE statement, but the statements use different methods for entering the digitized information into computer memory:

- **The DIGITIZE statement suspends program execution** while you position the plotter's pen to the desired location and waits until the [Enter] button is pressed on the plotter. The physical pen's coordinates and up/down status are read into computer memory only when the [Enter] button is pressed. When the computer receives the digitized information, program execution continues. Here is an example that uses the DIGITIZE statement:

  ```
  DIGITIZE Xvar,Yvar,PenStatus
  ```

  where Xvar and Yvar are the coordinates of the point that was plotted and PenStatus is the pen status which tells whether the pen is in the up or down position.

- **The CURSOR statement does not suspend program execution.** The physical pen's coordinates and up/down status are read into the specified variables immediately — without pressing the plotter's Enter button. Here is an example that uses the CURSOR statement:

  ```
  CURSOR Xvar,Yvar,PenStatus
  ```

  where Xvar and Yvar are the variables that receive the coordinates of the point that was plotted and PenStatus is the pen status which tells whether the pen is in the up or down position.

NOTE

The PenStatus parameter does not reflect the current pen's up/down status. Instead, it is set by the last drawing operation that affected pen status. For instance, if the last drawing operation was a MOVE, then the PenStatus parameter would indicate that the pen is "up". If the pen was put down since the MOVE (by a manual operation at the plotter), then the PenStatus parameter will not reflect the pen's up/down status correctly.

Keep in mind that the pen must be positioned at the desired location for digitizing prior to executing the CURSOR statement. The DIGITIZE statement allows you to position the pen and enter the digitized information after DIGITIZE is executed.

Digitizing the Logical Pen Location

The WHERE statement assigns the current logical pen coordinates and status to the specified variables. The parameters are the same as the parameters in the CURSOR and DIGITIZE statements.

```
WHERE x_variable,y_variable,pen_status_var
```

The location and up/down status of the logical pen is determined by the most recently executed statement which changes pen status or location. All of the plotting statements which direct pen movement also affect the logical pen location. In addition, statements and conditions which activate the default graphics conditions also lift the logical pen and move it to the origin (0,0). However, the physical pen's location and status are unaffected by activating the default graphics conditions.

The logical and physical pens often have the same location and status; any plotting statement which directs pen movement inside the current plotting area moves the physical pen as well as the logical pen.

The following program demonstrates the difference between the physical and logical pen positions as read by the CURSOR and WHERE statements. When program execution is suspended, move the pen (using the plotter's front panel controls) to a new location, lower the pen, execute CONT. The computer displays the resulting physical (CURSOR) and logical (WHERE) pen coordinate locations and pen status. In the example output below, the physical pen was moved to the coordinate location x = 76.6, y = 68.0, and lowered.

```
100 ASSIGN 7 to "hpgl,/dev/ohpgl"
110 PLOTTER IS 7 !    Specifies the plotting device.
120 MOVE 50,50 !      Moves the pen to the point (50,50)
130 PAUSE !           Pauses the program while you move
140 !                 the plotter pen to a new position.
150 WHERE WX,WY,WP !  Assigns logical pen position and
160 !                 status to the variables WX,WY,WP.
170 CURSOR CX,CY,CP ! Assigns physical pen position and
180 !                 status to the variables CX,CY,CP.
190 CLEAR
200 DISP USING "8A,2X,2(3D.D),3X,D";"WHERE",WX;WY;WP
210 DISP USING "8A,2X,2(3D.D),3X,D";"CURSOR",CX;CY;CP
220 END
```

The results from the programs execution are:

```
WHERE        50.0   50.0   0
CURSOR       76.6   68.0   1
```

Graphics Using HPGL Commands

To simplify communicating with the wide variety of HP graphics devices, a standard set of graphic commands has been adopted. The Hewlett-Packard Graphics Language (HPGL) consists of about sixty, two-letter commands that can be used to control the operation of most HP plotters. If fact, when BASIC statements are used to control an external plotter, they are converted (by the system) into a series of HPGL commands which are then sent to the plotter. Refer to the plotter's manual for details concerning which HPGL commands the plotter can recognize.

NOTE

In order to use these graphics techniques, you should have a firm grasp of plotting operations in general. In addition, you will need to determine which HPGL commands are supported on your plotter.

Also, you should avoid intermixing HPGL commands and Technical BASIC graphics commands, because BASIC makes some assumptions which you may have invalidated by using HPGL commands..

Introduction

While most plotting applications can be accomplished by using BASIC statements, some plotters have capabilities that can only be accessed by using HPGL commands. When it is neccessary (or desirable) to communicate directly with the plotter, the `OUTPUT` statement can be used to send HPGL commands. For example[1]:

```
ASSIGN 7 TO "hpib"
OUTPUT 705;"DF;";
```

Some Examples

This statement sends the HPGL command to restore the default conditions of the plotter. Many of the HPGL commands have one or more parameters. For instance:

```
ASSIGN 7 TO "hpib"
OUTPUT 705;"LT 6;";
```

This statement sets the line type to pattern number 6. The line type is plotter dependent and not likely to be the same pattern displayed on the CRT by the `LINE TYPE` statement in Technical BASIC.

[1] This must be a "raw" node—one with a primary address of **1f** in the minor number. See **ASSIGN** in the *Technical BASIC Reference Manual* for details.

In general, an HPGL command is terminated by either a semicolon or a line-feed. The parameters in commands are usually separated by commas. In the previous example statement, a semicolon is included in the string sent to the external plotter to indicate the end of a command. The `OUTPUT` statement's trailing semicolon suppresses the current end-of-line sequence from being sent to the plotter.

Some HPGL commands request the plotter to send back information to the computer. The **ENTER** statement is used to receive the information. The following example interrogates the plotter for the coordinates of the lower-left (P1) and upper-right (P2) graphics limit.

```
100 ! This program determines the coordinates of
105 ! the lower-left corner (P1) and upper-right corner (P2)
110 ! of the "plotting area" (in "absolute device units").
115 !
120 ASSIGN 7 TO "hpib"
130 !
140 ! Ask plotter to "Output Points P1 and P2".
150 OUTPUT 705; "OP;";
160 !
170 ! Now input the points.
180 ENTER 705 ; P1x,P1y,P2x,P2y
190 !
200 ! Now show the coordinates.
210 CLEAR @ DISP
220 DISP "Lower-left corner, P1: (";P1x;",";P1y;")"
220 DISP "Upper-right corner, P2: (";P2x;",";P2y;")"
230 END
```

The results of executing this program on an HP 7475A plotter are as follows:

```
Lower-left corner, P1: ( 250 , 596 )
Upper-right corner, P2: ( 10250 , 7796 )
```

The `OUTPUT` statement sends the "Output Points P1 and P2" command to the plotter, and the **ENTER** statement accepts the X and Y coordinates for P1 (lower-left corner) and P2 (upper-right corner) sent by the plotter. The values returned are in "absolute device units," not in GU's or UU's. One absolute device unit is equal to 0.025 millimetre.

You might wonder how the mapping of GU's, UU's, and absolute device units is accomplished. Consider the following statements[1]:

```
10 ASSIGN 7 TO "hpgl,/dev/ohpgl"
20 PLOTTER IS 7
```

This statement actually sends several HPGL commands to the plotter and accepts the current setting of P1 and P2 for use by the computer in converting the values used by Technical BASIC statements into the values needed for HPGL commands.

If you wish to change the locations of P1 and P2, it will be necessary to re-execute the `PLOTTER IS` statement (after changing P1 and P2). This allows Technical BASIC to become aware of the new graphics limits, and set up the correspondence between UU's (or GU's) and absolute device units.

[1] This must be an "auto-addressed" node—one with primary addressing in the minor number. See **ASSIGN** in the *Technical BASIC Reference Manual* for details.

Index

a

ABS 4-20
Absolute difference 4-11
Absolute graphics device units 15-83
ABSUM 4-71
Accessing data files 11-11
ACS 4-17
Activation record 6-34
Additional image specifiers 9-20
Algorithms 2-2, 2-6
Allocation of subprograms 6-12
ALPHA 9-6, 9-8, 9-11
Alpha screen 9-6
Alphanumeric inputs 9-24
AMAX 4-71
AMAXCOL 4-71
AMAXROW 4-71
AMIN 4-71
AMINCOL 4-71
AMINROW 4-71
Anisotropic scaling 15-23
Anticipating problems 9-25
Appending strings 5-5
Arbitrary loop exit points 3-22
AREAD 9-8, 9-30
Arithmetic:
- Hierarchy 4-5
- Operators 4-6

Array:
- Dimensions 4-24
- Displaying 4-27
- Empty 4-50
- Functions, misc. 4-71
- Numeric 4-22
- Printing 4-28
- Redimensioning 4-33

Scalar arithmetic 4–52
Storing 11–6
String 5–3, 5–17
Subscripts 4–23, 4–24
Summing rows and columns 4–54
Terminator 4–29
Transpose 4–56
Variable names 4–27

ASCII:
Characters 5–13, 11–10
File 2–23

ASN 4–17
Aspect ratio (width/height) 15–18, 15–34
ASSIGN 6–35, 9–6, 12–19, 13–24, 14–22, 15–9
ASSIGN# 11–13, 11–18, 11–20, 11–23, 11–25, 11–26

Assigning:
Array variables 4–27
String variables 5–2
Values to arrays 4–37
Variables 4–2

Assumptions, questioning 8–18
ATN 4–17
ATN2 4–17
Audio messages 9–4
AWRIT 9–8, 9–11
AXES 15–19
Axes intersection 15–20

b

Background color (raster) 15–54
BackSpace key 9–30
BASIC editor 2–14, 2–20
BASIC/DATA files 6–13, 11–12
BASIC/GRAF file 15–73
BASIC/PROG files 6–13, 6–14
BASIC/SUBP files 6–14, 6–16
BEEP 2–8, 9–4
BINAND 4–14

Binaries:
C 12–1
FORTRAN 14–1
Pascal 13–1

Binary:
- (base 2) 4–15
- Operations 4–14
- Programs 2–11

BINCMP 4–14
BINEOR 4–14
BINIOR 4–14, 4–14
BIT 4–14
Blank lines, displaying 9–12
Blocks 11–16
Boolean expressions 3–11, 4–10
Boundary conditions 7–3, 8–18
BPLOT 15–69
Branching:
- Conditions 8–19
- Event-initiated 3–24

BREAD 15–69
Break key 9–24, 9–28
Breaking programs up 3–30
Breakpoints 8–13
brt_pascalwrap procedure 13–5
BTD 4–16
Budget program, example 2–3
Buffer number 11–12
Bugs:
- Definition of 8–1
- Source of 8–3

Bulleted lists 1–6
Byte plotting (graphics) 15–69
Byte reading (graphics) 15–69, 15–69

C

C binaries:
- Compiling 12–3, 12–7, 12–8
- Error trapping 12–6
- File I/O 12–5, 12–19
- Introduction 12–1
- Linking 12–3, 12–7, 12–8
- makebin_c script 12–7, 12–8
- Maximum number of 12–5
- Parameter matching 12–5

Parameter passing . . . 12-9
Passing by reference . . . 12-10
Passing by value . . . 12-10
Restrictions . . . 12-5
String length . . . 12-15
C functions . . . 12-2
C programs . . . 12-2
Calculations from keyboard . . . 8-14
CALL . . . 6-8
CALLBIN . . . 2-11, 12-4, 13-4, 14-4
Capabilities, graphics . . . 15-6
Capabilities of displays . . . 9-7
CAPS LOCK . . . 9-25
CAT . . . 11-10
cc command . . . 12-8
CEIL . . . 4-20
CFLAG . . . 6-31, 6-35
CHAIN . . . 3-30, 6-30, 6-33
Chaining programs . . . 3-30
Chapter previews . . . 1-6
Character sets, printer . . . 9-14
Character size (graphics) . . . 15-65
Characteristics of COM . . . 6-28
Choosing program segments . . . 3-14
CHR$ . . . 2-11, 5-13, 9-14
Circle . . . 15-24
CLEAR . . . 9-9, 9-10
Clearing:
Flags . . . 6-31
The screen . . . 9-9
CLIP . . . 15-36, 15-42, 15-42, 15-47
Clock . . . 10-2
Closing files . . . 11-12, 11-31
CNORM . . . 4-71
CNORMCOL . . . 4-71
Code walk-throughs . . . 8-4
Codes, for keys . . . 9-30
Coding programs . . . 2-2, 2-8
COL . . . 4-29
Color pens . . . 15-54

Column-major order 4–31
COM 3–30, 4–26, 5–2, 6–27, 6–33
COM characteristics 6–28
Commands 2–12
Comments 2–9, 3–34
Common:
  Storage 4–26, 6–27
  Variables 3–30
Communication:
  Between programs 3–31
  Program/subprogram 6–19
Comparisons, numeric 4–11, 7–5
Compiling C binaries 12–3
Compiling FORTRAN binaries 14–3, 14–8
Compiling Pascal binaries 13–3, 13–8
Computer/human interface 9–1
CON 4–33
Concatenating statements 2–9
Concatenation, string 5–4
Conditional:
  Branching 3–12
  Execution 3–10
  GOTO 3–12
Console 9–6
Constant:
  Matrices 4–40
  Numeric 4–5
Constructs, nesting 3–14
CONT 3–5, 8–15, 9–25
Context 3–6, 6–10, 6–35, 6–35
Continuing execution 8–15
Control:
  C 8–12
  Characters 9–14
Conversions:
  Lettercase 5–15
  Number-base 4–15
  String 5–11
  Time and date 10–3
Coordinate systems 15–12

Copying:
- Program segments 2–20
- Subarrays 4–43

`COS` 4–18
`COT` 4–18
`CREATE` 6–35, 9–6, 11–12, 11–14, 11–25
`CROSS` 4–62
Cross:
- Product 4–62
- References 8–5

`CRT IS` 6–35, 9–6
`CSC` 4–18
`CSIZE` 15–65
`CSUM` 4–54
Current working directory 11–10
`CURSCOL` 9–11
`CURSOR` 15–52, 15–80
Cursor:
- Location 9–11
- Positioning 9–10
- Turning on 9–12

`CURSROW` 9–11
Cyclic timer interrupts 10–6

d

`DATA` 3–34, 4–39, 11–4
Data:
- File access 11–11
- Files 11–9
- Items 11–10
- Pointer 11–6
- Pointer, moving 11–8
- Structures 2–2, 2–6
- Types 2–10, 4–3, 11–24, 11–28

Date 10–2
`DATE` 10–2
`DATE$` 10–2
Date format conversions 10–3
Debugging:
- Methods 8–4
- Programs 2–3, 2–24

Decisions . . . 3-1
Declarations:
  Implicit . . . 4-4
  Of function parameters . . . 6-7
Declaring:
  COM variables . . . 6-27
  Pass parameter types . . . 6-24
  Variables . . . 2-10, 4-4
DEF FN . . . 3-34
Default graphics limits . . . 15-12
DEFAULT OFF . . . 6-35, 7-2
DEFAULT ON . . . 6-35
Default scale . . . 15-14, 15-15
Defaults, graphics . . . 15-75
DEG . . . 4-17, 6-35, 15-63, 15-67
Delay interrupts . . . 10-6
DELETE . . . 2-16
Deleting program lines . . . 2-16
Describing the problem . . . 2-3
DET . . . 4-71
DETL . . . 4-71
dev directory . . . 15-9
Developing programs . . . 2-2
Device file . . . 15-9
Device selector . . . 9-6, 15-9, 15-82
Device units, absolute graphic . . . 15-83
Devices, graphics . . . 15-9
Devices, graphics output . . . 15-75
Difference:
  Absolute . . . 4-11
  Relative . . . 4-11
DIGITIZE . . . 15-4, 15-52, 15-79
Digitizing . . . 15-76
Digitizing graphics limits . . . 15-49
DIM . . . 2-10, 3-34, 4-4, 5-2, 5-17
Dimensioned length, string . . . 5-2
Dimensioning:
  Implicit . . . 4-26
  Numeric arrays . . . 4-24
  Strings . . . 5-2, 5-17

Directory . . . 11-10
DIRECTORY . . . 6-11, 6-12, 6-17
Disabling keys . . . 9-30
DISP . . . 2-8, 9-6, 9-10, 9-12
DISP item separators . . . 9-16
DISP USING . . . 4-31
Display:
  Capabilities . . . 9-7
  Screen . . . 9-6
  Screen modes . . . 9-10
Displaying:
  Blank lines . . . 9-12
  Messages . . . 9-5
DIV . . . 4-8
Documenting programs . . . 2-3, 2-24
Documents:
  External . . . 2-29
  Internal . . . 2-34
DOT . . . 4-71
DRAW . . . 6-35, 15-56
Drawing . . . 15-56
DTB$ . . . 4-15
DTH$ . . . 4-15
DTO$ . . . 4-15
DTR . . . 4-18
DUMP ALPHA . . . 2-14, 9-13
DUMP GRAPHICS . . . 2-14, 15-74
Dyadic operators . . . 4-8

e

Editing:
  Global operations . . . 2-14
  Search operations . . . 2-18
Editor:
  BASIC . . . 2-14
  vi . . . 2-22
Elements of BASIC program . . . 2-8
Ellipse . . . 15-23
Empty arrays . . . 4-50
ENABLE KBD . . . 9-24, 9-28
Enabling keys . . . 9-28

END 2–9
`END` 2–9
End-of-record marker 11–13
Entry point 12–4, 13–4, 14–4
EOF:
  Conditions 11–29
  Markers 11–20, 11–24, 11–29
EOL, sequence 9–10
EOR:
  Conditions 11–29
  Markers 11–13, 11–20, 11–29
`EPS` 4–20
`ERRL` 7–7
`ERRM` 7–8
`ERRN` 7–7
Error:
  Default response 7–2
  File 11–29
  Handling 7–1
  Location 7–7
  Messages 2–14, 7–8
  Numbers 7–7
  Reporting 7–2
  Trapping 7–6, 9–25
Escape-code sequences 9–15, 9–30
Euclidian norm 4–72
Evaluating:
  Numeric expressions 4–4
  Strings 5–4
Event-initiated branching 3–24
Events, types of 3–24
`EXP` 4–20
Expressions:
  Calls and functions 4–9
  Evaluating numeric 4–4
  String 5–4, 5–18
Extending `BASIC/DATA` files 11–23
Extensible files 11–23
External documents 2–29

f

fc command 14–9
Field boundaries, DISP and PRINT 9–16
Field specifiers 9–17
File access:
  Random 11–24, 11–26
  Serial 11–11
  text/data 11–31
File I/O:
  C binaries 12–5, 12–19
  FORTRAN binaries 14–6, 14–22
  Pascal binaries 13–6, 13–24
File:
  ASCII 2–23
  BASIC/DATA 6–13
  BASIC/GRAF 15–73
  BASIC/PROG 6–13, 6–14
  BASIC/SUBP 6–14, 6–16
  Buffers 11–18
  Closing 11–12, 11–31
  Data types 11–28
  Names 6–14
  Opening 11–31
  Overhead 11–17
  Pointer 11–18
  Selector 9–6, 9–13
  Size calculations 11–14
  text/data 11–31, 12–5, 13–6, 14–6, 14–22
FINDPROG 6–11, 6–13
f_init subroutine 14–4
FLAG 6–31, 6–35
Flags, system 6–30
FLIP 6–35
FLOOR 4–20
FN END 3–34
FNORM 4–71
Formal parameters 6–4, 12–2, 13–2, 14–2
Formatted printing 9–15
FOR..NEXT 3–18

FORTRAN binaries:
- Compiling . . . 14–3, 14–8
- Error trapping . . . 14–7
- File I/O . . . 14–6, 14–22
- Introduction . . . 14–1
- Linking . . . 14–3, 14–8
- `makebin_c` script . . . 14–8, 14–9
- Maximum number of . . . 14–6
- Parameter matching . . . 14–5
- Parameter passing . . . 14–10
- Passing by reference . . . 14–10
- Passing by value . . . 14–10
- Restrictions . . . 14–6
- String length . . . 14–18

FORTRAN programs . . . 14–2
FORTRAN SUBROUTINEs . . . 14–2
`FP` . . . 4–20
Froebenius norm . . . 4–72
Functions:
- Binary . . . 4–14
- C . . . 12–2
- Constant . . . 6–3
- Data-type declarations . . . 6–7
- Limitations of user-defined . . . 6–7
- Local variables . . . 6–5
- Misc. array . . . 4–71
- Misc. numeric . . . 4–20
- Multi-line . . . 6–5
- Passing parameters to . . . 6–4
- Resident . . . 2–11
- Resident trig . . . 4–17
- Step . . . 4–10
- String . . . 5–9, 5–14
- Timer . . . 10–9
- User-defined . . . 2–11, 5–16, 6–1, 6–2

g

`GCLEAR` . . . 9–9, 15–3
General steps in development . . . 2–2
`GET` . . . 2–22, 6–13
Getting Started Guide . . . 1–3, 2–14

GLOAD 15–73
Global:
  Declarations 6–35
  Program editing 2–14
GOSUB 3–7
GOTO:
  Conditional 3–12
  Unconditional 3–6
GRAD 4–17, 6–35
Graphics 15–1
Graphics capabilities 15–6
Graphics character size 15–65
Graphics, clearing screen 9–9
Graphics defaults 15–75
Graphics devices, specifying 15–9
Graphics, digitizing 15–49, 15–52
Graphics dump 15–74
Graphics examples 15–2, 15–4
Graphics, initializing 15–9
Graphics input 15–4, 15–52
Graphics limits 15–13, 15–36, 15–83
Graphics limits, default 15–12
Graphics limits, digitizing 15–49
Graphics limits, moving 15–14, 15–32, 15–36
Graphics limits, range 15–33
Graphics limits, summary 15–47
Graphics mapping 15–12, 15–14
Graphics output devices 15–9, 15–75, 15–75
Graphics plotters 15–11
Graphics printers 15–74
Graphics scaling 15–14
Graphics tablets 15–11
Graphics units, absolute 15–83
Graphics Units (GU's) 15–14, 15–15, 15–29
Graphics windows 15–11
GRID 15–21
GSTORE 15–73

h

Halting program execution 3–3
Hardcopy, of the screen 2–14
Hardware installation 1–2
Hewlett-Packard Graphics Language (HPGL) 15–82
Hexadecimal (base 16) 4–15
Hiding the details 2–6
Hierarchy:
  Arithmetic 4–5
  String 5–4
HIL input locators 15–11
`HMS` 10–4, 10–9
`HMS$` 10–3
HP-IB graphics tablets 15–11
HP-IB primary address 15–82
HP-UX Technical BASIC Getting Started Guide 1–3, 2–14
HP-UX Technical BASIC Language Reference 1–3
HP-UX:
  File system 1–2
  Knowledge 1–2
HPGL 15–82
`HTD` 4–16
Human/computer interface 9–1, 9–3

i

Identity matrix 4–42
`IDN` 4–33
`IDRAW` 15–56
`IMAGE` 4–31, 9–17
Image specifiers:
  Additional 9–20
  Definition 9–17
  Numeric 9–18
  String 9–19
`IMOVE` 15–56
Implicit:
  Declarations 4–4
  Dimensioning 4–26
  Redimensioning 4–35
Incremental drawing 15–56

Incremental moving 15–56
Incremental plotting 15–56
Indenting program lines 2–28
`INF` 4–20
`INIT` 3–32, 8–15, 8–16
Initializing graphics 15–9
`INPUT` 2–8, 9–24, 11–4
Input, graphics 15–4, 15–52
Input locators 15–11
Input:
  Alphanumeric 9–24
  From keyboard 9–21, 9–23
  Keyboard 9–30
Insert mode (screen) 9–10
Inserting program lines 2–15
`INT` 4–13
`INTEGER` 2–10, 3–34, 4–3, 11–24
Integral numbers 2–10
Interactions between timers and subprograms 10–10
Internal documents 2–24
Interrupts:
  Cyclic timer 10–6
  Delay 10–6
  Time-of-day 10–8
  Timer 10–6
Intersection of axes 15–20
Interval timing 10–5
Intrinsic functions:
  General 2–11
  String 5–9, 5–14
Inverting matrices 4–64
`IP` 4–21
`IPLOT` 15–56
Isotropic scaling 15–24
Iterations 3–18

j

Joining strings 5–5
Jump 3–6

k

KEY LABEL . . . 9–23
Key:
  Buffer . . . 9–30
  Codes . . . 9–30
  Disabling . . . 9–30
  Enabling . . . 9–28
  Labels . . . 3–26
  Special function . . . 9–22
Keyboard:
  Calculations . . . 8–14
  Enable mask . . . 9–28
  Errors . . . 7–2
  Input . . . 9–30
  Inputs . . . 9–21, 9–23
Keywords . . . 1–5, 2–8

l

LABEL . . . 15–65
Label area (softkeys) . . . 3–26
Label direction . . . 15–65
Label origin . . . 15–65
Labeling . . . 15–65
LAXES . . . 15–14
LBND . . . 4–71
ld command . . . 12–8, 13–9, 14–9
LDIR . . . 15–65
LEN . . . 5–9
Length of string . . . 5–2, 12–15, 13–19, 14–18
LET . . . 2–8, 4–2
Lettercase conversion . . . 5–15
LGT . . . 4–21
Libraries, subprogram . . . 6–10
LIMIT . . . 15–17, 15–32, 15–36, 15–37, 15–38, 15–39, 15–40, 15–47
Limiting range of values . . . 4–12
Limits, graphics . . . 15–13
Limits, physical (graphics) . . . 15–12
Line length (LINE TYPE) . . . 15–64
LINE TYPE . . . 15–64

Line:
  Labels 2-9, 2-25
  Numbers 2-9
  Where referenced (XREF L) 8-6
Linear:
  Equations 4-66
  Flow 3-3
Linking C binaries 12-3
Linking FORTRAN binaries 14-3, 14-8
Linking object files 12-8, 13-9, 14-9
Linking Pascal binaries 13-3, 13-8
LINPUT 9-24, 11-4
Listings 2-14
Lists:
  In program lines 2-8
  Parameter 6-4, 6-20
LOAD 6-13
LOADBIN 12-4, 13-4, 14-4
Loading subprograms 6-17, 6-18
Local:
  Declarations 6-35
  Variables 6-6
LOCATE 15-14, 15-36, 15-37, 15-38, 15-39, 15-40, 15-47
Locators, input 15-11
LOG 4-21
Logging in and out 1-2
Logical pen location 15-76, 15-80
Logical record 11-10, 11-16, 11-25
Loop counter 3-18
Looping 3-18
LORG 15-65
LWC$ 5-15

m

Machine language programs 2-11
makebin_c script 12-3, 12-7, 12-8
makebin_f script 14-3, 14-8, 14-9
makebin_p script 13-3, 13-8, 13-9
Manual:
  Organization 1-4
  Overview 1-1

Mapping, graphics 15–12
Markers:
  EOF 11–13, 11–20, 11–24, 11–29
  EOR 11–13, 11–20, 11–29
MASS STORAGE IS 6–35
Mass storage tutorial 11–9
MAT 4–39, 4–52
MAT DISP 4–28
MAT DISP USING 4–31
MAT INPUT 4–37
MAT PRINT 4–28
MAT PRINT USING 4–31
MAT READ 4–39
MAT..CON 4–33, 4–40
MAT..CROSS 4–62
MAT..CSUM 4–54
Math hierarchy 4–6
MAT..IDN 4–33, 4–42
MAT..INV 4–64
Matrix:
  Inversion 4–64
  Multiplication 4–57
MAT..RSUM 4–54
MAT..SYS 4–66
MAT..TRN 4–56
MAT..ZER 4–33, 4–40
MAX 4–12, 4–21
MAXAB 4–71
MAXABCOL 4–72
MAXABROW 4–72
MDY 10–5
MDY$ 10–4
Mechanics of program development 2–14
Memory management (subprograms) 6–34
Menu (softkeys) 3–26
MERGE 2–21
Messages:
  Accepting 9–21
  Audio 9–4
  Displayed 9–5
  From operator 9–21
  To the operator 9–4

Methods of passing parameters . . . 6–21
Millimetre scaling . . . 15–26
`MIN` . . . 4–12, 4–21
`mknod` command . . . 15–9
`MOD` . . . 4–8
Model:
  Of computer/human interface . . . 9–2
  Of software design process . . . 8–3
Modes, display screen . . . 9–10
Modules, Pascal . . . 13–2
Moment . . . 4–62
Monadic operators . . . 4–8
Monochromatic pens . . . 15–54
`MOVE` . . . 15–56
Moving graphics limits . . . 15–14, 15–32, 15–36
Moving the pen . . . 15–56
Moving:
  Data pointer . . . 11–6
  Program segments . . . 2–20
`MSCALE` . . . 15–22, 15–47
Multiplying matrices . . . 4–57

n

Names:
  Array variables . . . 4–27
  Of C binaries . . . 12–4
  Of FORTRAN binaries . . . 14–4
  Of numeric variables . . . 4–2
  Of Pascal binaries . . . 13–4
  String variables . . . 5–2
  Subprogram . . . 6–14
Nesting constructs . . . 3–14
Newline character (C) . . . 12–20
Non-executed statements . . . 3–34
`NORMAL` . . . 6–35, 8–5, 9–10
Notation for program lines . . . 2–9
`NPAR` . . . 6–25
Null character (C string terminator) . . . 12–16
`NUM` . . . 4–21, 5–11
Number-base conversions . . . 4–15

Numbers:
- INTEGERs 4-3
- Random 4-19
- Range of 4-3
- REAL 4-3
- SHORT 4-3

Numeric:
- Arrays 4-22
- Comparisons 4-11
- Data types 4-3
- Functions, misc. 4-20
- Image specifiers 9-18
- Variables 4-4

O

Object files 12-8, 13-9, 14-9
Octal (base 8) 4-15
OFF CURSOR 6-35, 9-12
OFF ERROR 6-35
OFF KEY# 3-27, 6-35
OFF KYBD 6-35
OFF TIMEOUT 6-35
OFF TIMER# 6-35, 10-6, 10-9
ON CURSOR 6-35, 9-12
ON ERROR 6-35, 7-6, 9-25, 11-29
ON KEY# 3-24, 6-35, 9-22
ON KYBD 6-35, 9-24, 9-30
ON TIMEOUT 6-35
ON TIMER# 6-35, 10-6
ON..GOSUB 3-15
Opening files 11-31
Operations, string 5-18
Operator:
- Arithmetic 4-6
- Errors 7-3
- String 5-5

OPTION BASE 2-13, 4-4, 5-17
Optional pass parameters 6-25
Organization of manual 1-4
Origin 15-32

Origin, graphics 15–14
`OTD` 4–16
Output devices, graphics 15–9, 15–75
Overview 1–1, 1–6

p

P1, P2 15–83
Parameter matching:
  C binaries 12–5
  FORTRAN binaries 14–5
  Pascal binaries 13–5
Parameter passing:
  C binaries 12–9
  FORTRAN binaries 14–10
  Pascal binaries 13–10
Parameter:
  Formal 6–4, 12–2, 13–2, 14–2
  Lists 6–4, 6–20
  Optional 6–25
  Pass 6–4, 12–2, 13–2, 14–2, 14–4
  Passing 6–4, 6–19
  Statement 2–8
Pascal binaries:
  Compiling 13–3, 13–8
  Error trapping 13–7
  File I/O 13–6, 13–24
  Introduction 13–1
  Linking 13–3, 13–8
  `makebin_p` script 13–8, 13–9
  Maximum number of 13–6
  Parameter matching 13–5
  Parameter passing 13–10
  Passing by reference 13–10
  Passing by value 13–10
  Restrictions 13–6
  String length 13–19
Pascal modules 13–2
Pascal procedures 13–2
Pascal programs 13–2

Pass parameters:
- C Binaries 12–2, 12–4
- Declaring types 6–24
- FORTRAN Binaries 14–2, 14–4
- Optional 6–25
- Pascal Binaries 13–2, 13–4
- Types 6–20

Passing:
- By reference 6–21, 12–10, 13–10, 14–10
- By value 6–23, 12–10, 13–10, 14–10
- Parameters 6–4, 6–19, 6–21

`PAUSE` 3–5, 8–13
`PAUSE` 9–28
Pausing execution 8–12
`pc` command 13–9
`PDIR` 15–63
`PEN` 6–35, 15–54
Pen control (`PLOT`) 15–56
Pen, logical 15–76, 15–80
Pen, physical 15–76
Pen status 15–53, 15–79
Physical limits (graphics) 15–12
Physical pen location 15–76
Physical records 11–16
`PI` 4–21
Pitches, beeper 9–5
Pixels (picture elements) 15–69
`PLOT` 6–35, 15–56
Plotter considerations 15–75
`PLOTTER IS` 6–35, 15–9, 15–47, 15–75
Plotters 15–11
Plotting 15–56
Plotting area 15–12, 15–32, 15–36
Plotting bounds 15–12, 15–32, 15–36, 15–42
Plotting bounds, digitizing 15–49
Plotting bounds (summary) 15–47
Plotting devices 15–75
Plotting direction 15–63
Plotting pixels 15–69
Plotting with HPGL 15–82
Pointer, data 11–6

POS 4–21, 5–9
Positioning cursor 9–10
Pre-run 3–32
Precedence, arithmetic 4–6
Prerequisites 1–2
Primary address 15–82
PRINT 9–13
PRINT# 11–12, 11–18, 11–25
PRINT ALL 6–35, 8–5, 9–10
PRINT item separators 9–16
PRINT USING 4–31, 9–17
Printer character sets 9–14
Printer graphics 15–74
PRINTER IS 6–35, 9–10, 9–13
Printers 9–13
Printing:
  Arrays 9–16
  Formatted 9–15
  Screen contents 2–14
Problem solving steps 2–2
Procedures, Pascal 13–2
Program segments:
  Choosing 3–14
  Repeating 3–18
Program:
  Binary 2–11
  C 12–2
  Communication between 3–31
  Counter 3–2
  Definition of 2–9
  Editing globally 2–14
  Elements 2–8
  Entering 2–14
  Execution 3–32
  Flow 3–1, 3–2
  FORTRAN 14–2
  Line numbers 2–9
  Lines 2–9
  Lines, maximum length 2–9
  Listings 2–14
  Machine language 2–11

Pascal . . . 13-2
Running . . . 2-14
Storing . . . 2-14
Structure . . . 3-1
Program/subprogram communication . . . 6-19
Prohibited statements (in IF..THEN) . . . 3-12
Put (vi) . . . 2-23

q

Questioning assumptions . . . 8-18
Quotes in strings . . . 11-5

r

RAD . . . 4-17, 6-35
Random:
File access . . . 11-11, 11-24, 11-26
Numbers . . . 4-19
RANDOMIZE . . . 4-19
Range limits . . . 4-12
Raster images, retrieving . . . 15-73
Raster images, storing . . . 15-73
RATIO . . . 15-18, 15-34
Ratio, aspect (width/height) . . . 15-18, 15-34
READ . . . 11-4
READ# . . . 11-13, 11-18, 11-24, 11-26
Reading pixels . . . 15-69
Reading:
Flags . . . 6-31
Text from screen . . . 9-30
READTIM . . . 10-9
REAL . . . 2-10, 3-34, 4-3, 11-24
Real numbers . . . 2-10
Record size calculations . . . 11-14
Records . . . 11-16
Redimensioning arrays . . . 4-33
Refining . . . 2-2, 2-7
Reflecting images . . . 15-48, 15-51
Relative difference . . . 4-11
Relative plotting . . . 15-56
Relocatable object files . . . 12-8, 13-9, 14-9
REM . . . 2-25, 3-34

Remark statements . . . 2–25
REN . . . 2–16
Renaming variables . . . 2–19
Renumbering programs . . . 2–16
Repeat . . . 3–20
Repeat factor (LINE TYPE) . . . 15–64
Repeating strings . . . 5–14
Repetition . . . 3–17
REPLACEVAR . . . 2–19
Reset key . . . 9–24, 9–28
Resident:
  Binary functions . . . 4–14
  Functions . . . 2–11
  Trig functions . . . 4–17
RESTORE . . . 11–8
Restrictions:
  C binaries . . . 12–5
  FORTRAN binaries . . . 14–6
  Pascal binaries . . . 13–6
Retrieving raster images . . . 15–73
RETURN . . . 3–7, 7–7
REV$ . . . 5–14
Reversing strings . . . 5–14
RMD . . . 4–21
RND . . . 4–19
RNORM . . . 4–72
RNORMCOL . . . 4–72
RNORMROW . . . 4–72
ROTATE$ . . . 4–14
Rotating lines . . . 15–56
Rounding . . . 4–13
ROW . . . 4–30
Row-major order . . . 4–33
RPLOT . . . 15–56
RPT$ . . . 5–14
RSUM . . . 4–54
RTD . . . 4–18
RUN . . . 3–32, 6–17, 8–15, 8–16
Run-time errors . . . 7–3
Running programs . . . 2–14

S

SAVE . . . 2–23, 6–11, 6–16
Scalar array arithmetic . . . 4–52
SCALE . . . 15–14, 15–22, 15–37, 15–38, 15–39, 15–40, 15–47
Scaling, anisotropic . . . 15–23
Scaling, default graphic . . . 15–15
Scaling, graphics . . . 15–14
Scaling (graphics) . . . 15–22
Scaling, isotropic . . . 15–24
Scaling, millimetre . . . 15–26
Scaling (summary) . . . 15–47
SCAN . . . 2–18
Scanning for literals . . . 2–18
SCRATCH . . . 6–11
SCRATCHBIN . . . 12–4, 13–4, 14–4
SCRATCHSUB . . . 6–17, 6–34
Screen:
  Clearing . . . 9–9
  Display . . . 9–6
  Dumps . . . 2–14, 15–74
  Reading text from . . . 9–30
Screenwidth . . . 9–7
Searching for literals . . . 2–18
SEC . . . 4–18
Seed, random numbers . . . 4–19
Selection of program segments . . . 3–10
Selector:
  Device . . . 9–6
  File . . . 9–6
Self-documenting programs . . . 2–25
Sending messages . . . 9–4
Separators, DISP and PRINT . . . 9–16
Serial file access . . . 11–11, 11–13, 11–20, 11–23
Service routines . . . 3–26, 9–30
SETGU . . . 15–29, 15–47
Setting flags . . . 6–31
SETUU . . . 15–29, 15–47
SFLAG . . . 6–31, 6–35
SGN . . . 4–21
SHORT . . . 2–10, 3–34, 4–3, 11–24
SHOW . . . 15–22, 15–47

Simple:
- Branching 3–6
- Strings 5–3

SIN 2–11, 4–18
Single-stepping programs 8–16
SINGLESTEP 8–16
Size:
- of files 11–15
- of records 11–16

Softkeys 3–25, 9–22, 9–28
Software:
- Installation 1–2
- Testing 8–17

Solving:
- Problems 2–2
- Simultaneous equations 4–66

Spaghetti code 3–6
Special (device) file 15–9
Special function keys 9–22, 9–28
Specifiers:
- Field 9–17
- Image 9–17

Specifying graphics devices 15–9
SQR 4–21
Starbase type 15–9
Statements 2–8
Step functions 4–10
Stepwise refinement 2–2, 2–7
STOP and END 3–4
Stopwatch example 9–23
Storage, common 4–26
STORE 2–21, 6–11, 6–15
Storing raster images 15–73
Storing:
- Arrays 11–7
- Data in variables 11–3
- Programs 2–14

String length 12–15, 13–19, 14–18
String terminator (C) 12–16

String:
- Alphanumeric 2–10
- Arrays 5–3, 5–17
- Concatenation 5–4
- Conversions 5–11
- Definition of 5–2
- Dimensioning 5–17
- Evaluating expressions 5–4
- Expressions 5–18
- Functions 5–9, 5–14
- Hierarchy 5–4
- Image specifiers 9–19
- In numeric expressions 4–10
- Length 5–9
- Operators 5–5
- Position 5–9
- Repeat 5–14
- Reverse 5–14
- Simple 5–3
- Subscripts 5–6
- Trim 5–15
- Variable length 5–2
- Variable names 5–2

SUB 2–11, 6–8, 6–11

Subarrays, copying 4–42, 4–47

SUBEND 6–8, 6–11

SUBEXIT 6–11

Subprogram:
- Benefits of 6–9
- Creating 6–11
- Definition 2–11
- Introduction to 6–1, 6–8
- Libraries 6–10
- Loading 6–17, 6–18
- Memory management 6–34
- Names 6–14
- Scratching 6–17

Subprogram/program communication 6–19

Subroutine:
- General suggestions 3–9
- GOSUB 3–7

SUBROUTINEs, FORTRAN 14–2
Subscript:
  Bounds 4–24
  String 5–6
Substring:
  Definition 5–6
  Double-subscript 5–7
  Position 5–9
  Single-subscript 5–6
`SUM` 4–72
Summing arrays 4–54
Syntax of keywords 1–5
`SYS` 4–66
System:
  Clock 10–2
  Error message 7–8
  Flags 6–30
  Of equations 4–66
  Timers 10–9
  Warnings 7–3

t

`TAB` 9–12, 9–16
Tablets, graphics 15–11
Tablets, HP-IB graphics 15–11
`TAN` 4–18
Terminator, numeric array 4–29
Testing:
  Programs 2–3, 2–24
  Software 8–17
`text/data` files 11–31, 12–9, 13–10, 14–10, 14–22
Tick marks 15–19
Tilde “ ” character 11–5
`TIME` 10–2
`TIME$` 10–2
Time format conversions 10–3
Time of day 10–2
Time-of-day interrupts 10–8

Timer:
- Functions . . . 10–9
- Interrupts . . . 10–6
- Interrupts (w/ subprograms) . . . 10–10
- Using . . . 10–6

Timing intervals . . . 10–5
Tones . . . 9–5
Top-down design . . . 6–10
TRACE . . . 8–8
TRACE ALL . . . 8–11
TRACE VAR . . . 8–9
Tracing:
- All flow . . . 8–11
- Branches . . . 8–7
- Variables . . . 8–10

Transposing arrays . . . 4–56
Trapping errors . . . 7–6, 9–25, 11–29
Trigonometric functions . . . 4–17
TRIM$ . . . 5–15
Trimming strings . . . 5–15
TRN . . . 4–56
TYP . . . 11–13, 11–28
Type fields (files) . . . 11–28
Type:
- Fields (files) . . . 11–13
- Of pass parameters . . . 6–21
- Of program flow . . . 3–2
- Pass parameters . . . 6–24

Types of lines . . . 15–64

u

UBND . . . 4–72
UNCLIP . . . 15–45, 15–47
Unconditional **GOTO** . . . 3–6
Understanding the problem . . . 2–3
Unit matrix . . . 4–42
UNIX . . . 1–2
Unlinked object files . . . 12–8, 13–9, 14–9
UPC$ . . . 5–15, 9–25
User documents (for your programs) . . . 2–29
User Units (UU's) . . . 15–22, 15–29

User-defined:
- Function limitations . . . 6–7
- Functions . . . 2–11, 5–16, 6–1, 6–2
- Keys . . . 3–25

Using plotters . . . 15–75

V

`VAL` . . . 4–21, 5–11
`VAL$` . . . 4–21, 5–12
Variable:
- Allocation of . . . 3–32
- Assigning . . . 4–2
- Declarations . . . 2–10
- In `COM` . . . 6–27
- Names . . . 2–25, 4–2
- Names of strings . . . 5–2
- Numeric . . . 4–4
- Numeric arrays . . . 4–22
- Renaming . . . 2–19
- String . . . 5–2
- String length . . . 5–2
- Types of . . . 2–10
- Where used (`XREF V`) . . . 8–5

Vector:
- Components . . . 4–63
- Cross product . . . 4–62
- Magnitude . . . 4–63

vi editor . . . 2–22

W

Walk-throughs . . . 8–4
Warnings . . . 7–3
`WHERE` . . . 15–80
While . . . 3–21
Width of screen . . . 9–7
Windows, graphics . . . 15–11

X

XREF 8–5
XREF L 8–6
XREF V 8–5

Y

Yank (vi) 2–23

Z

ZER 4–33
Zero matrices 4–40

MANUAL COMMENT CARD

HP-UX Technical BASIC

Programming Guide, Vol. 2

for HP 9000 Computers

Manual Reorder No. 97068-90001

Name: ______________________________

Company: ______________________________

Address: ______________________________

____

Phone No: ______________

Please note the latest printing date from the Printing History (page ii) of this manual and any applicable update(s); so we know which material you are commenting on ______________.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT
PACKARD

HP Part Number
97068-90001

Microfiche No. 97068-99001
Printed in U.S.A. 2/86

97068-90606
For Internal Use Only