BASIC 5.0/5.1 Graphics Techniques

HP 9000 Series 200/300 Computers

HP Part Number 98613-90032

HEWLETT PACKARD

Hewlett-Packard Company

3404 East Harmony Road, Fort Collins, Colorado 80525

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

WARRANTY

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

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.

January 1987...Edition 1

November 1987...Edition 2. This edition reflects the 5.0 corrections and 5.1 additions.

Table of Contents

Chapter 7: Utility Routines

Appendix A

Introduction to Graphics 1

Introduction to Graphics 1

Welcome

One of the most exciting features of your Series 200/300 computer is its graphics capability. Graphic messages are often a better way of communicating information using primarily non-textual information.

Prerequisites

This manual introduces you to the powerful set of graphics statements in the BASIC Programming Language, as well as teaches you how to orchestrate them to produce pleasing output. This manual assumes you have read chapters 1 through 5 of the *BASIC Programming Techniques* manual, and that you will refer to that manual when you encounter any programming topics you do not understand.

If you have a question as to what binary to load for a particular keyword or option, see the *BASIC Language Reference.* Note that you must load the BIN files named GRAPH and GRAPHX before you can enter most graphics statements. You may need to load other BIN files, depending on what computer you have. Refer to the *Installing and Maintaining the BASIC System* for information about the BIN files. Finally, certain programs in this manual require BIN files such as MAT that you might not readily associate with graphics.

You may have a disc called *Manual Examples Disc.* The part numbers vary, depending on what disc drive you have, but the disc contents are identical. The *Manual Examples Disc* disc contains programs which may be helpful, but they are not overemphasized in this manual.

Why Graphics?

Below is some data. As quickly as you can, determine if its overall trend is steady, rising or falling. Are there any periodic motions to it? If so, how many cycles are represented in the one hundred points?

Table 1-1. Lexically Represented Set of Voltage Data

Time (sec.)	Voltage	Time (sec.)	Voltage
1	16.10	51	16.69
2	16.25	52	16.55
3	16.25	53	16.65
4	16.28	54	16.62
5	16.36	55	16.67
6	16.31	56	16.68
7	16.27	57	16.81
8	16.08	58	16.88
9	16.10	59	16.87
10	16.06	60	17.07
11	16.07	61	17.16
12	16.17	62	17.16
13	16.14	63	16.94
14	16.26	64	16.98
15	16.34	65	16.83
16	16.40	66	16.83
17	16.56	67	16.71
18	16.60	68	16.81
19	16.44	69	16.83
20	16.51	70	16.84

Table 1-1. Lexically Represented Set of Voltage Data (continued)

Time (sec.)	Voltage	Time (sec.)	Voltage
21	16.35	71	16.81
22	16.41	72	16.98
23	16.28	73	17.05
24	16.19	74	17.23
25	16.30	75	17.30
26	16.24	76	17.34
27	16.27	77	17.14
28	16.44	78	17.22
29	16.44	79	17.16
30	16.57	80	16.96
31	16.60	81	17.02
32	16.70	82	16.99
33	16.72	83	16.84
34	16.66	84	17.06
35	16.58	85	16.96
36	16.62	86	17.15
37	16.46	87	17.30
38	16.33	88	17.37
39	16.34	89	17.39
40	16.36	90	17.51
41	16.45	91	17.32
42	16.52	92	17.47
43	16.56	93	17.29
44	16.77	94	17.17
45	16.89	95	17.10
46	16.80	96	17.07
47	16.96	97	17.06
48	16.80	98	17.09
49	16.74	99	17.13
50	16.77	100	17.20

Below is a graph of the data in the preceding table. Observe that the graphical nature of the output makes it much clearer what the data is doing. This clarity and understandability at a glance is what computer graphics is all about. Many example programs are included in the pages that follow. Type in and run them as you progress from simply drawing a jagged line to creating complex graphics.

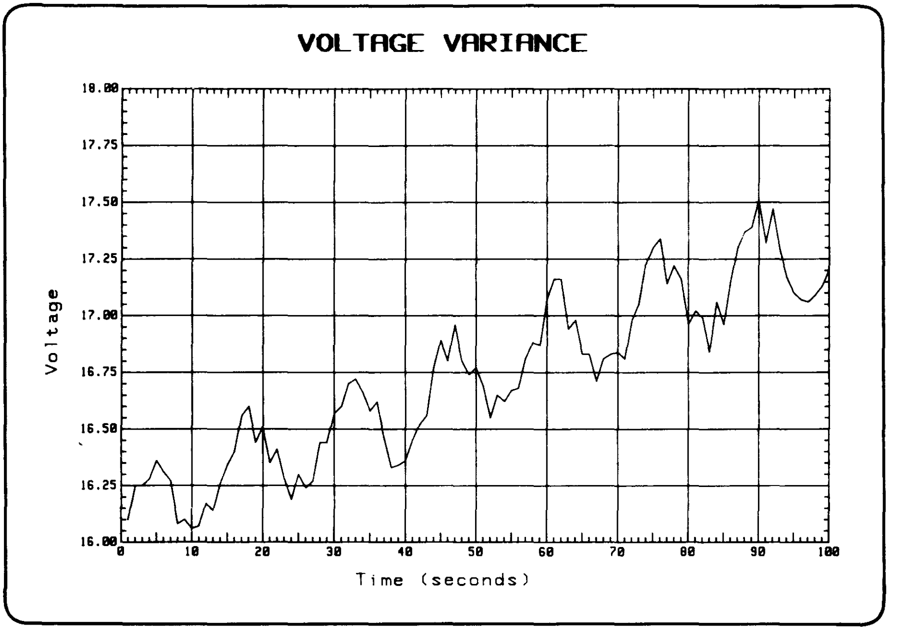


Figure 1-1. Graphic Representation of Data Is Often More Useful

Drawing Lines

To draw lines, you can simply say PLOT, followed by the X and Y coordinates of the point you want to draw a line to. The following program does just that.

```
10    GINIT                       ! Initialize various graphics parameters.
20    PLOTTER IS CRT,"INTERNAL"   ! Use the CRT screen
30    GRAPHICS ON                 ! Turn on the graphics screen
40    FOR X=2 TO 100 STEP 2       ! Points to be plotted...
50      PLOT X,RND+50             ! Get a data point and plot it against X
60    NEXT X                      ! RND returns a value between 0 and 1
70    END                         !
```

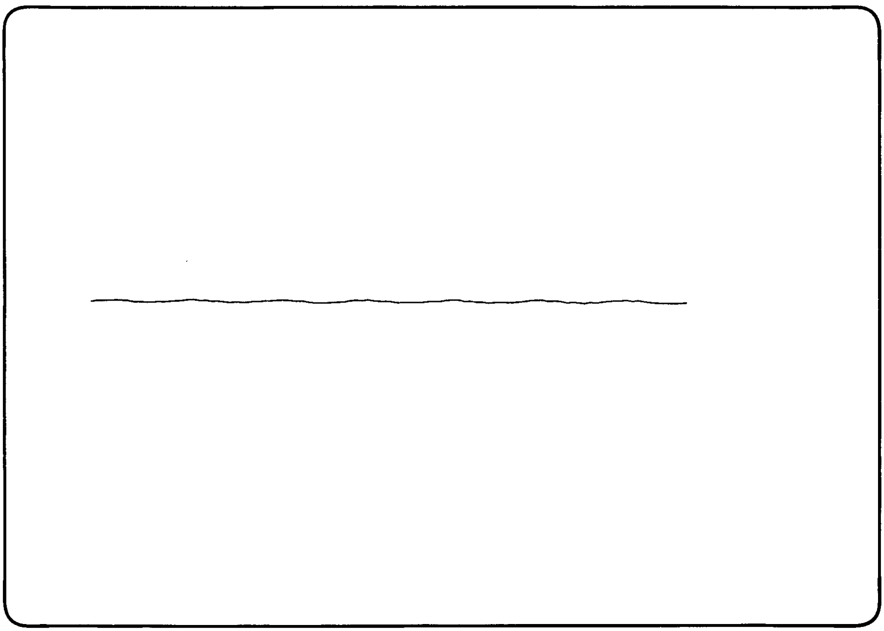


Figure 1-2. Example of Plotting Random Data

As you can see, this simple seven-line program is all you need to draw a simple plot. Granted, it would be nice to know what we are plotting, and what the units are, etc., but we'll get there in due time.

The GINIT statement on line 10 means Graphics **Init**ialize. This is almost always the first graphics statement you would execute. As its name implies, it sets various graphics parameters to their default values, and it is a shorthand way of executing up to fourteen other statements (see the *BASIC Language Reference* manual for details).

The GRAPHICS ON statement on line 30 allows you to see what the program is drawing if you have separate alpha and graphics. On bit-mapped displays, graphics and alpha are always on, unless you have modified the display mask. More on this later.

Line 50 contains the heart of the program. In a loop, the PLOT statement draws to each successive point, which is determined by the loop control variable `X` for the X direction and the value returned by the function `RND+50` for the Y direction. The constant, 50, is used to center the line on the screen so it is not displayed in your softkey display area.

Scaling

Probably the first reaction you had when looking at the previous plot was "That doesn't show me anything...." That's true; it doesn't show much information. There is not enough variation in the curve; it's too straight to show us anything. If we exaggerated the Y direction to the point where we could see the variations, the line would better represent the plotted data.

This problem can be remedied by scaling. In this context, scaling is "defining the values the computer considers to be at the edges of the plotting surface." By definition, the left edge is the minimum X, the right edge is the maximum X, the bottom is the minimum Y, and the top is the maximum Y. Thus any point you plot which has X and Y coordinates within these ranges will be visible.

Two statements are available to define your own values for the edges of the plotting surface. The first one we'll deal with is SHOW, which forces X and Y units to be equal. This is called **isotropic** scaling, and it is often desirable. For example, when drawing a map, you will probably want one mile in the east-west direction to be the same size as a mile in the north-south direction. Here is an example of SHOW:

```
SHOW 0,100,16,18
```

This causes the plotting area to be defined such that there is a rectangle in that plotting area whose minimum X is 0, maximum X is 100, minimum Y is 16, and maximum Y is 18, *using isotropic units.* As mentioned above, isotropic means that one unit in the X direction is equal to one unit in the Y direction. Hence, if the plotting area were square, the above SHOW statement would define the minimum X to be 0, maximum X to be 100, minimum Y to be -33 (not 16) and maximum Y to be 67 (not 18). The reason for this is that since we have to have X and Y units identical, the SHOW statement centers the specified area in the plotting area, allowing whatever extra room it needs to ensure that the rectangle is completely contained in the plotting area. There will be extra room in either the X or Y direction, but not both.

Since you were defining unit sizes with the SHOW statement, you were working with User-Defined Units (UDUs). Both the SHOW statement and the WINDOW statement (covered next) specify user-defined units.

The next example uses a SHOW statement to define the edges of the screen to appropriate values. The X values used in the SHOW statement (0 and 100) come from the facts that there are 100 data points and that axes are more meaningful when the origin is at zero and not one. The Y values (for this type of plot) must be determined either by you or by the computer itself. We are using a random number function to simulate data being received from some device.

If you want the computer to determine the X maximum, you could do it this way:

```
210    Xmax=-1.0E308                    ! Smaller than smallest value in array
210    FOR I=1 TO N                     ! N is the number of elements in array
220      IF X(I)>Xmax THEN Xmax=X(I)
230    NEXT I
```

A similar set of program lines could be used to determine the minimum value of X.

The MAX and MIN functions provide an alternate method for determining the maximum and minimum values of a set of numbers. However, these functions require the use of the MAT binary. Below is an example of how these functions are used.

```
110    Ymin=INT(MIN(Y(*)))
120    Ymax=MAX(Y(*))
130    Ymax=INT(Ymax)+(Ymax<>INT(Ymax))
```

Line 110 calculates the "floor" of the minimum value in an array of Y values. The floor of a number is the **greatest integer less than or equal to** that number, i.e., rounding *down* to the nearest integer. Lines 120 and 130 calculate the "ceiling" of the maximum value in the array of Y values. The ceiling of a number is the **smallest integer greater than or equal to** that number, i.e, rounded *up* to the nearest integer.

Back to our example, the Y values being used (16 and 18) come from the RND function. In real applications, you probably will not know beforehand what the range of the data will be, in which case you can use the method described above.

```
10    GINIT                        ! Initialize various graphics parameters.
20    PLOTTER IS CRT,"INTERNAL"    ! Use the internal screen
30    GRAPHICS ON                  ! Turn on the graphics screen
40    SHOW 0,100,15,19             ! Isotropic scaling: left, right, bottom, top
50    FOR X=2 TO 100 STEP 2        ! Points to be plotted...
60      PLOT X,RND+17              ! Get a data point and plot it against X
70    NEXT X                       ! RND returns a value between 0 and 1
80    END                          !
```

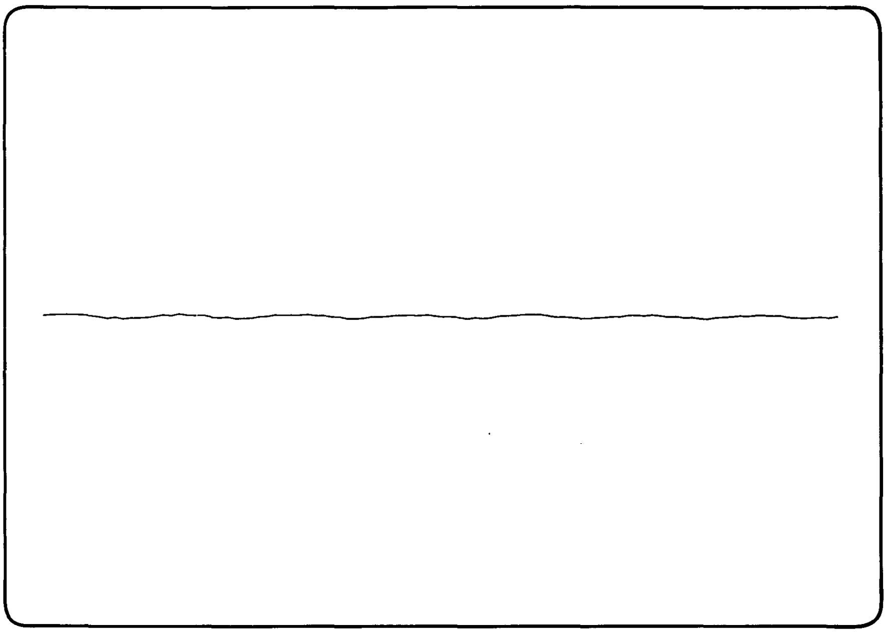


Figure 1-3. Better Use of X-Axis Scaling

As you can see, the SHOW statement takes care of centering the curve on the screen, but since the range of X values is so much larger than the range of Y values (0 to 100 versus 16 to 18), it still does not give us enough resolution to see what the data is doing. Isotropic scaling is desirable in many cases. In many other cases, however, it is not. In this example, we are hypothetically graphing the voltage from a sensor versus time, and it makes no sense at all to force seconds to be just as "long" as volts. Since we are dealing with data types which are not equal, it would be better to use unequal, or **anisotropic**, scaling. We can use the other scaling statement: WINDOW. This will not force X units to be equal to Y units. Instead, the scaling is determined by the axis range.

```
10    GINIT                       ! Initialize various graphics parameters.
20    PLOTTER IS CRT,"INTERNAL"   ! Use the internal screen
30    GRAPHICS ON                 ! Turn on the graphics screen
40    WINDOW 0,100,15,19          ! Anisotropic scaling: left, right, bottom, top
50    FOR X=2 TO 100 STEP 2       ! Points to be plotted...
60      PLOT X,RND+17             ! Get a data point and plot it against X
70    NEXT X                      ! RND returns a value between 0 and 1
80    END                         !
```

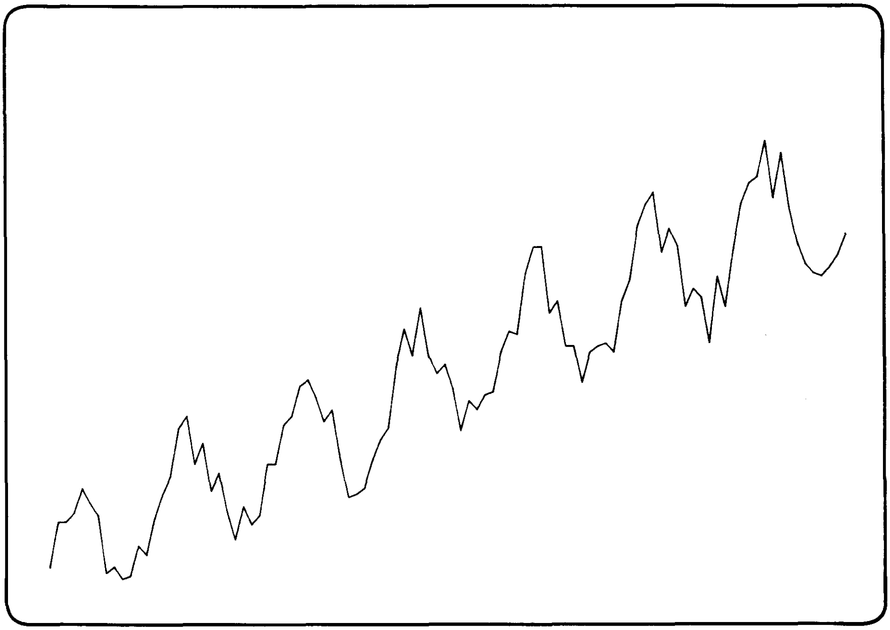


Figure 1-4. Better Use of Y-Axis Scaling

This plot looks much better than the last one; we can easily see variations in the data. To test how the Y axis range, 15-19, affects relative variations in the data, change `WINDOW 0,100,15,19` to `WINDOW 0,100,30,50` and change `RND+17` to `RND+40`. Run the program again and note that the line is less ragged (remember that RND ranges between 0 and 1).

There is still one problem, though. We can see *relative* variations in the data, but what are the units being used? That is, is the height of the curve signifying differences of microvolts, millivolts, megavolts, dozens of volts, or what? And we probably wouldn't want the text (explaining units, etc.) to be written in the same area that the curve is in, as it could obstruct part of the data curve. Therefore, we need to be able to specify a subset of the screen for plotting the curve; put explanatory notes outside this area. The next section tells you how to do this.

Defining a Viewport

A viewport is a subset of the plotting area. This is called the **soft clip area**, and it is delimited by the **soft clip limits**. Clip, because any line segments which attempt to go outside these limits are cut off at the edge of the subarea. Soft, because we can override these limits by turning off the clipping with the CLIP OFF statement (more about this later). There are **hard clip limits** also, and these are defined to be the absolute limits of the plotting area. Under no circumstances can a line be drawn outside of these limits. There is no way to override the hard clip limits, as we can with soft clip limits.

GDUs and UDUs

Before we define a viewport, we need to know about the two different types of units which exist. These two types of units are **UDUs (User-Defined Units)** and **GDUs (Graphics Display Units)**. In order for viewports to be predictable, they must always be specified in the same units. Since UDUs are subject to change by the user, GDUs are used when specifying the limits of a VIEWPORT statement. GDUs are fixed for the CRT, so a viewport is associated with the screen, rather than the graphical model used in your program.

Unless you specify otherwise, the screen (but *not* necessarily an external plotter) is considered to have the following expanse:

- In the Y direction (the shorter side), units are 0 through 100.
- In the X direction, the range of scaling units depends on the aspect ratio of the screen (the aspect ratio is determined by executing the RATIO command); here are some examples:
  - 0 through 133.444816054 (on the Models 216 and 226)
  - 0 through 131.362467866 (for the Models 236 and 236C)
  - 0 through 128.0701754 (for the 98542A and 98543A)
  - 0 through 133.3767927 (for the 98544, 98545, 98546, 98547, and 98700)

These are GDUs. The length of a GDU is defined as "One percent of the shorter edge of the plotting area." There are some important characteristics of GDUs which you should know:

- The lower left of the plotting area is *always* 0,0.
- GDUs are isotropic; that is, one unit in the X direction is the same distance as one unit in the Y direction.

Since the height of the screen is shorter than the width of the screen, the shorter edge is in the Y direction, therefore, Ymax in GDUs is 100. If the screen had been higher than it is wide, Xmax in GDUs would have been 100. That was the easy part. Once you've decided which edge is shorter, and thus defined the units along that edge, you need to find out how many GDUs in extent the *longer* sides are. This will be covered in detail in the "Using Graphics Effectively" chapter. For now, we'll just observe that the GDU limits are as mentioned above.

Specifying the Viewport

The VIEWPORT statement defines the extent of the soft clip limits in GDUs. It specifies a subarea of the plotting surface which acts just like the entire plotting surface except you can draw outside the subarea if you turn off clipping (more about clipping later). The VIEWPORT statement in the following program specifies that the lower left-hand corner of the soft clip area is at 10,15 and the upper right-hand corner is at 120,90. This is the area which the WINDOW statement affects. Also note line 50; the FRAME statement. This draws a box around the current soft clip limits. It is used in this example so you can see the area specified by the VIEWPORT statement.

```
10    GINIT                      ! Initialize various graphics parameters.
20    PLOTTER IS CRT,"INTERNAL"  ! Use the internal screen
30    GRAPHICS ON                ! Turn on the graphics screen
40    VIEWPORT 10,120,15,90      ! Define subset of screen area
50    FRAME                      ! Draw a box around defined subset
60    WINDOW 0,100,15,19         ! Anisotropic scaling: left,right,bottom,top
70    FOR X=2 TO 100 STEP 2      ! Points to be plotted...
80      PLOT X,RND+17            ! Get a data point and plot it against X
90    NEXT X                     ! RND returns a value between 0 and 1
100   END                        !
```

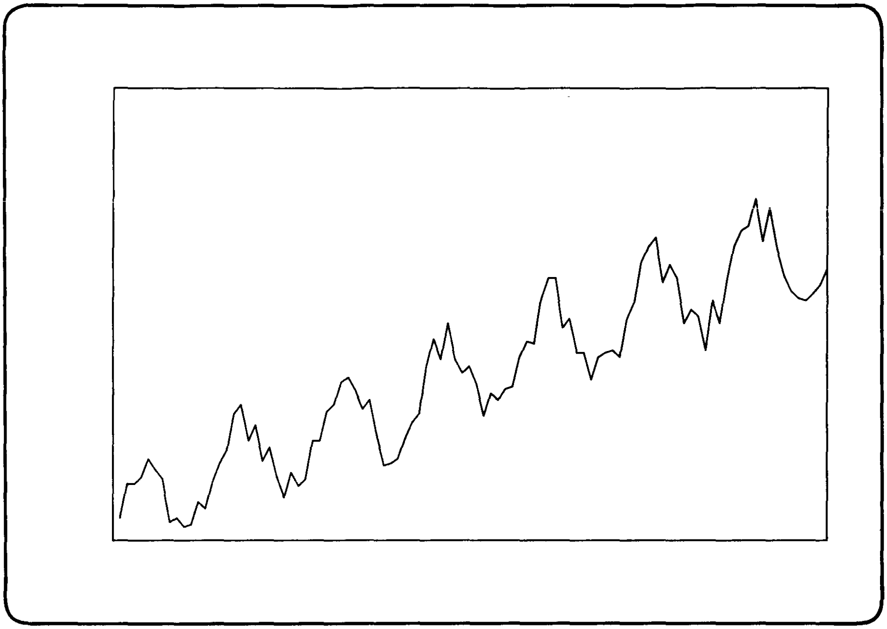


Figure 1-5. Using VIEWPORT to Define Soft-Clip Limits

Labeling a Plot

With the inclusion of the VIEWPORT statement, we have enough room to include labels on the plot. Typically, in a Y-vs.-X plot like this, there is a title for the whole plot centered at the top, a Y-axis title on the left edge, and an X-axis title at the bottom.

There is a statement called LABEL which writes text onto the graphics screen. You can position the label by using a MOVE or PLOT statement to get to the point at which you want the label to be placed. The lower left corner of the label is at the point to which you moved. In other words, move to the position on the screen at which the lower left corner of the text is placed. Note that the LORG statement will move you to the lower left corner of the label. (The relative origin for labels can be changed with the LORG statement.)

Notice in the following plot that the Y-axis label on the left edge of the screen is created by writing one letter at a time. We only need to move to the position of the first character in that label because each label statement automatically terminates with a carriage return/linefeed. This causes the pen to go one line down, ready for the next line of text. (There is a better way to plot vertical labels; we'll see it in the next chapter.)

```
10    GINIT                         ! Initialize various graphics parameters.
20    PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
30    GRAPHICS ON                   ! Turn on the graphics screen
40    MOVE 45,95                    ! Move to left of middle of top of screen
50    LABEL "VOLTAGE VARIANCE"      ! Write title of plot
60    MOVE 0,65                     ! Move to center of left edge of screen
70    Label$="Voltage"              ! Write Y-axis label
80    FOR I=1 TO 7                  ! Seven letters in "Voltage"
90      LABEL Label$[I,I]           ! Label one character
100   NEXT I                        ! et cetera
110   MOVE 45,10                    ! X: center of screen; Y: above key labels
120   LABEL "Time (seconds)"        ! Write X-axis label
130   VIEWPORT 10,120,15,90         ! Define subset of screen area
140   FRAME                         ! Draw a box around defined subset
150   WINDOW 0,100,16,18            ! Anisotropic scaling: left/right/bottom/top
160   FOR X=2 TO 100 STEP 2         ! Points to be plotted...
170     PLOT X,RND+16.5             ! Get a data point and plot it against X
180   NEXT X                        ! et cetera
190   END                           ! finis
```

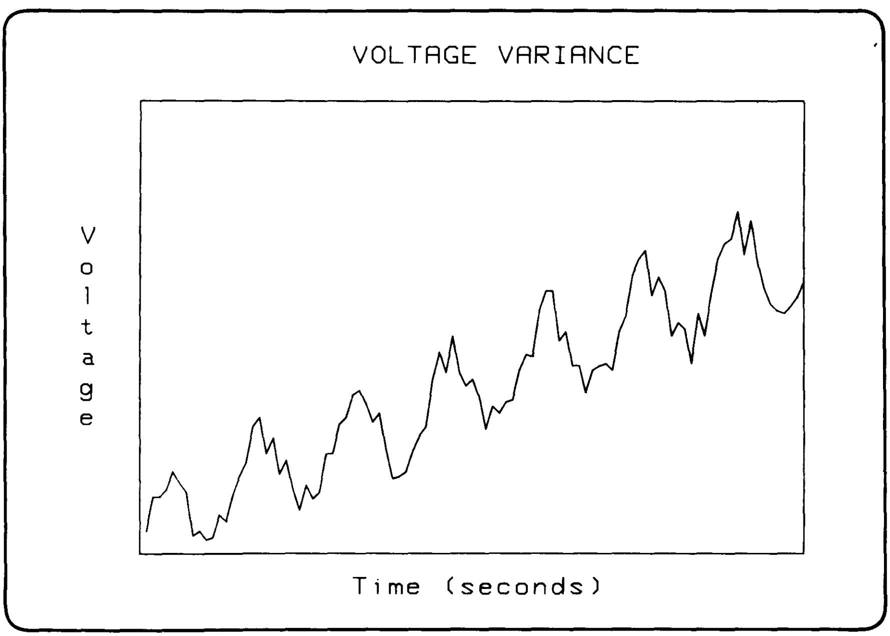


Figure 1-6. Labeled Plot

Now we know what we are measuring—voltage vs. time—but we still do not know the units being used. What we need is an X-axis and a Y-axis, and they need to be labelled with numbers in appropriate places. The AXES statement fits the bill here.

Axes and Tick Marks

The AXES statement draws X and Y axes and short lines, perpendicular to the axes, to indicate the spacing of units. These short lines are called **tick marks**. The axes may intersect at any desired point; it need not be the actual origin—the point 0,0. The tick marks may be any distance apart, and you can select the "major tick count" for each axis. The major tick count is the total number of tick marks drawn for every large one. This makes it convenient to count by fives or tens or whatever you chose the major tick count to be. And finally, you can specify how long you want the major tick marks to be. This is measured in GDUs. Insert the `AXES` statement in your program and rerun it to see the difference.

```
10    GINIT                         ! Initialize various graphics parameters.
20    PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
30    GRAPHICS ON                   ! Turn on the graphics screen
40    MOVE 45,95                    ! Move to left of middle of top of screen
50    LABEL "VOLTAGE VARIANCE"      ! Write title of plot
60    MOVE 0,65                     ! Move to center of left edge of screen
70    Label$="Voltage"              ! Write Y-axis label
80    FOR I=1 TO 7                  ! Seven letters in "Voltage"
90      LABEL Label$[I,I]           ! Label one character
100   NEXT I                        ! et cetera
110   MOVE 45,10                    ! X: center of screen; Y: above key labels
120   LABEL "Time (seconds)"        ! Write X-axis label
130   VIEWPORT 10,120,15,90         ! Define subset of screen area
140   FRAME                         ! Draw a box around defined subset
150   WINDOW 0,100,16,18            ! Anisotropic scaling: left/right/bottom/top
160   AXES 1,.1,0,16,5,5,3          ! Draw X- and Y-axes with appropriate ticks
170   FOR X=2 TO 100 STEP 2         ! Points to be plotted...
180     PLOT X,RND+16.5             ! Get a data point and plot it against X
190   NEXT X                        ! et cetera
200   END                           ! finis
```

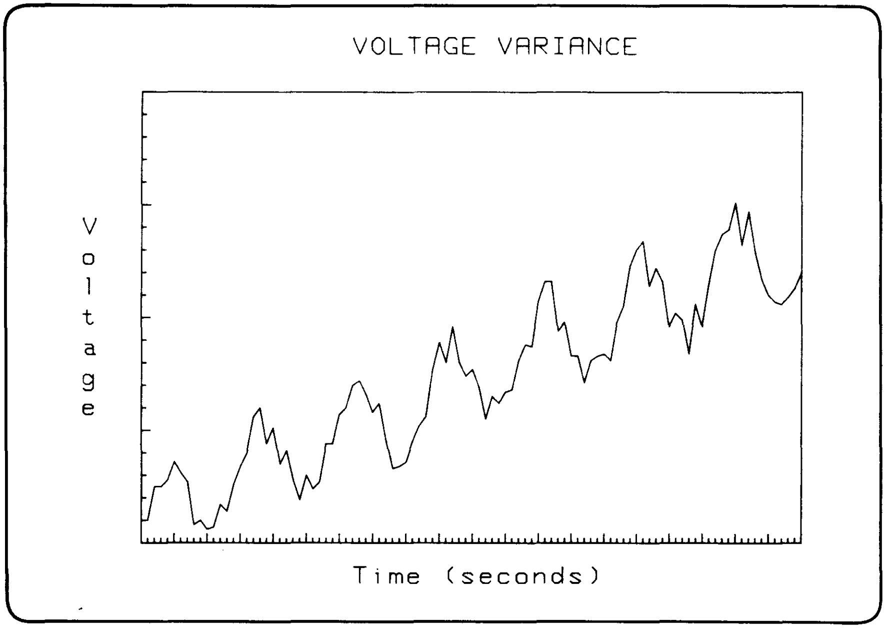


Figure 1-7. Plot with Axes and Tick Marks

Line **160** of the program contains the AXES statement and its parameters. The parameters are explained as follows:

`1,.1` are the respective values for the X and Y axes tick-mark spacings. The X axis has 1 display unit between tick marks, and the Y axis has .1 display unit between tick marks, in current display units.

`0,16` are the respective values used to determine the origin of the Y and X axes. The Y axis crosses the X axis at X equals 0. The X axis crosses the Y axis at Y equals 16.

`5,5` specify the x-major and y-major counts, respectively. These counts are the number of "minor" (shorter) ticks between "major" (longer) tick marks on the axes. The value of 5 specifies that there will be a major tick mark every 5th tick mark.

`,3` specifies the major tick size, in graphics display units. The value of 3 specifies that the major ticks are 3 GDU's long (the default is 2).

This chapter has shown you how easy it is to write a program whose output is in graphical form. Now you have the basic knowledge needed to get into graphics in a serious way. The next chapter discusses these statements in greater depth, so you can to make even more effective graphical output.

Using Graphics Effectively 2

Using Graphics Effectively 2

In the last chapter we discussed the more elementary graphics operations. In this chapter, we will discuss how to use those statements more fluently, and introduce additional graphics statements.

The *Manual Examples* disc, which was shipped with this manual, contains programs found in this chapter. If you have the disc, load the appropriate program and run it. Otherwise, it is beneficial to take time to type in the listed programs and run them. Either way, experiment with them until you are familiar with the demonstrated concepts and techniques.

Note

Some programs in this chapter require the MAT (matrix) BIN file.

More on Defining a Viewport

Recall that the VIEWPORT statement defines a subset of the screen in which to plot. More precisely, the VIEWPORT statement **defines a rectangular area into which the WINDOW coordinates will be mapped.** (If you didn't catch that, don't panic. It will become clearer.) VIEWPORT immediately rescales the plotting area; thus, it is a good programming practice to follow every VIEWPORT statement with a WINDOW statement. The VIEWPORT also invokes clipping at its edges. There will be more about clipping later in this chapter.

The Y direction edge values default to 0 through 100 in Y. The X direction left edge value is 0. The right edge value can vary depending on what computer you have (approximately 128-133). Technically, these are UDSs (User-Defined Units), although default UDUs are equivalent to the GDUs until you change the UDUs with a SHOW or a WINDOW. The length of a GDU is defined as "One percent of the shorter edge of the plotting area." To recap the important characteristics of GDUs:

- The lower left of the plotting area is 0,0.
- GDUs are isotropic; that is, one unit in the X direction is the same distance as one unit in the Y direction.

As we mentioned in the last chapter, it is trivial to determine how long the shorter edge of screen is in GDUs, but substantially more involved to calculate the length of the longer edge in GDUs. Since the height of the screen is shorter than the width of the screen, the shorter edge is in the Y direction; therefore, Ymax in GDUs is 100. If the screen had been higher than it is wide, *X*max in GDUs would have been 100. Now for the interesting part.

Remember that GDUs are isotropic: X and Y units are the same length. This means that the length in GDUs of the longer edges of the plotting surface is closely related to the **aspect ratio** of the plotting surface. The aspect ratio is the ratio of width to height of the plotting surface. There is a function called RATIO which returns the quotient of these values. Thus, if the plotting area is wider than it is high, RATIO returns a value greater than one. If the plotting area is higher than it is wide, RATIO returns a value less than one, and if the plotting area were perfectly square, RATIO would return 1. To try this, type

`RATIO` [Return] or [ENTER]

The returned value is something like 1.33376792699. This lets you know how the X-direction maximum range compares with the Y-direction maximum range.

Using this function, we can derive two formulas which are almost indispensible when writing a general-purpose VIEWPORT statement:

```
X_gdu_max=100*MAX(1,RATIO)
Y_gdu_max=100*MAX(1,1/RATIO)
```

These two statements define the maximum X and maximum Y in GDUs. *This will work no matter what plotting device you are using.* Now that we have `X_gdu_max` and `Y_gdu_max` defined, we have complete control of the subset we want on the plotting surface. Suppose we want:

- the left edge of the viewport to be 10% of the hard clip limit width from the left edge,
- the right edge of the viewport to be 1% of the hard clip limit width from the right edge,
- the bottom edge of the viewport to be 15% of the hard clip limit height from the bottom, and
- the top edge of the viewport to be 10% of the hard clip limit height from the top.

We would specify:

```
VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
```

Now, armed with this new knowledge, let's return to the program which defined the viewport, and update the VIEWPORT statement accordingly. You may load this program from file "SinViewprt" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                  ! Clear the alpha display
110   GINIT                         ! Initialize various graphics parameters.
120   PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
130   GRAPHICS ON                   ! Turn on the graphics screen
140   X_gdu_max=100*MAX(1,RATIO)    ! How many GDUs wide the screen is
150   Y_gdu_max=100*MAX(1,1/RATIO)  ! How many GDUs high the screen is
160   VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
! Define subset of screen area
170   FRAME                         ! Draw a box around defined subset
180   WINDOW 0,100,16,18            ! Anisotropic scaling: left/right/bottom/top
190   FOR X=2 TO 100 STEP 2         ! Points to be plotted...
200     PLOT X,RND+16.5             ! Get a data point and plot it against X
210   NEXT X                        ! RND returns a value between 0 and 1
220   END
```

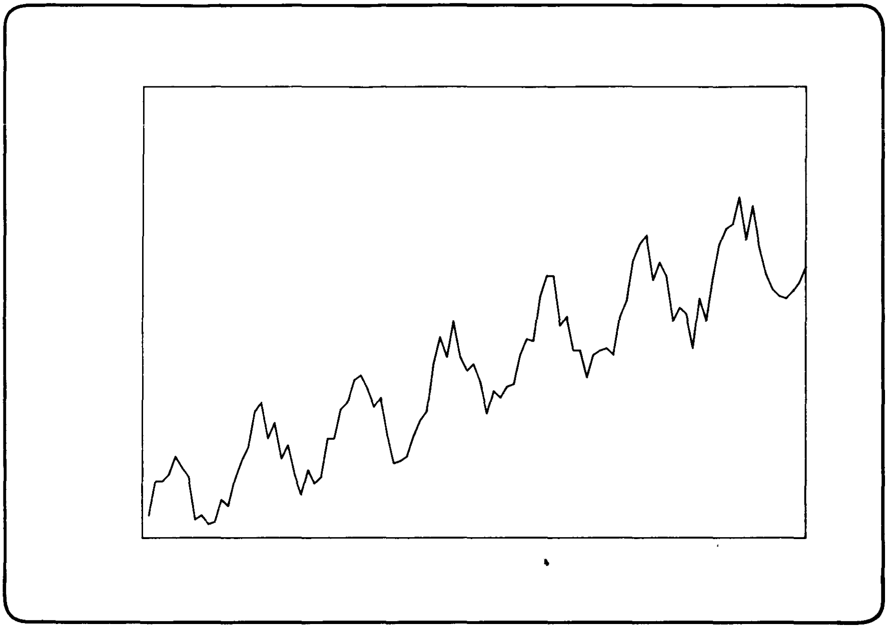


Figure 2-1. General-Purpose VIEWPORT (SinViewprt)

More on Labelling a Plot

There are three statements which complement the LABEL statement.

The first is CSIZE, which means *c*haracter *size.* CSIZE has two parameters: character cell height (in GDUs) and aspect ratio. The height measures the character *cell* size. A character cell contains a character and some blank space above, below, left of, and right of the character. This blank space allows packing character cells together without making the characters illegible. The amount of blank space depends, of course, on which character is contained in the cell. Focus on CSIZE in the program. Other statements are described later.

This small program shows how the CSIZE statement changes the size of characters. You may load this program from file "Csize" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                  ! Clear the alpha display
110   DIM Text$[50]                 ! Allow the long strings
120   GINIT                         ! Initialize various graphics parameters
130   PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
140   GRAPHICS ON                   ! Turn on the graphics screen
150   FRAME                         ! Draw a box around the screen
160   WINDOW -1,1,10,1              ! Anisotropic units
170   LORG 4                        ! Bottom center of labels is ref. pt.
180   FOR I=1 TO 6                  ! Six labels total
190     READ Csize,Text$            ! Read the characters cell size and text
200     CSIZE Csize                 ! Use Csize
210     MOVE 0,SQR(I)*3+1           ! Move to appropriate place
220     LABEL Text$                 ! Write the text
230   NEXT I                        ! Looplooplooplooplooploop
240   DATA 30,T,20,his,10,isjustlike,7,thosecutelittlecharts
250   DATA 5,thatyoualwaysseeinyourfriendly
260   DATA 3,neighborhoodoptometristsoropticiansoffice.
270   END
```

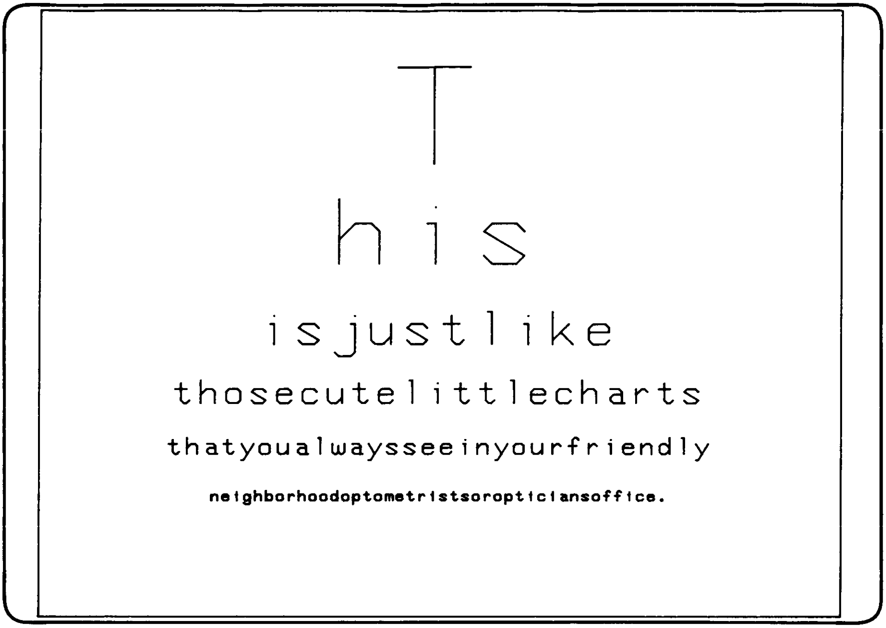


Figure 2-2. Changing Graphics Character Size

The FOR..NEXT loop writes lines of text on the screen with different character sizes. The DATA statements contain both pieces of information. Incidentally, notice also the WINDOW statement. It specifies a Ymin *larger* than the Ymax. This causes the top of the screen to have a lesser Y-value than the bottom. This is perfectly legal.

The next program deals with the relationship between the size of the character, per se, and the size of the **character cell**—that rectangle in which the character is placed. This program is on file "CharCell" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                      ! Clear the alpha display
110   GINIT                             ! Initialize various graphics parameters
120   PLOTTER IS CRT,"INTERNAL"         ! Use the internal screen
130   GRAPHICS ON                       ! Turn on the graphics screen
140   FRAME                             ! Draw a box around the screen
150   SHOW 0,36,-7.5,22.5               ! Isotropic units; Left/Right/Bottom/Top
160   FOR X=0 TO 36                     ! \
170     FOR Y=0 TO 15                   !  \
180       MOVE X-.1,Y+.1                !   \
190       DRAW X+.1,Y-.1                !    \
200       MOVE X+.1,Y+.1                !     > Draw all the little Xs.
210       DRAW X-.1,Y-.1                !    /
220     NEXT Y                          !   /
230   NEXT X                            !  /
240   FOR I=0 TO 27 STEP 9              ! \
250     CLIP I,I+9,0,15                 !  \
260     FRAME                           !   > Draw boxes around the character cells
270   NEXT I                            !  /
280   CLIP OFF                          ! Deactivate clipping so LABELs will work
290   CSIZE 50                          ! Character cells half the screen high
300   MOVE 0,0                          ! Starting point (LORG 1 by default)
310   LABEL "AbCd"                      ! Sample letters
320   CSIZE 7,.45                                    ! \
330   LORG 6                                         !  \
340   MOVE 18,22                                     !   > Write the title
350   LABEL "Size of Character in Character Cell"    !  /
360   END
```

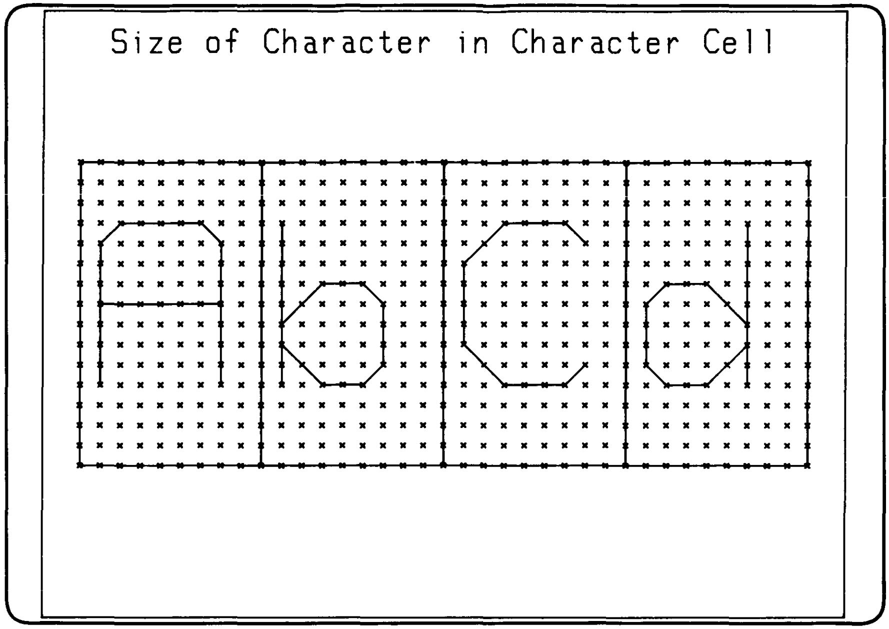


Figure 2-3. Character Cells (CharCell)

As the diagram shows, a character is drawn inside a rectangle, with some space on all four sides. The character's height and width are measured in GDUs and are specified by the CSIZE statement. Program lines **250** through **280** subdivide the rectangle into four 9 wide by 15 high grids. Characters are drawn in this framework, called the **symbol coordinate system**. Of course, the little Xs in the plot above are not drawn when you label a string of text; they are there solely to show the position of the characters within the character cell.

Again, character cell height is measured in GDUs, and the definition of aspect ratio for a character is identical to the definition of aspect ratio for the hard clip limits mentioned earlier: the width divided by the height. Thus, if you want short, fat letters, use an aspect ratio of 1.5 or larger. If you want tall, skinny letters, use an aspect ratio less than 0.5.

`CSIZE 3` Cell 3 GDUs high, aspect ratio 0.6 (default).

`CSIZE 6,.3` Cell 6 GDUs high, aspect ratio 0.3 (tall and skinny).

`CSIZE 1,2` Cell 1 GDU high, aspect ratio 2 (short and fat).

Note that you do not have to specify a second parameter (the aspect ratio) in the CSIZE statement. This defaults to 0.6.

The second statement you need is LORG, which means label origin. This lets you specify which point on the label ends up at the point moved to before writing the label. You may load the following program from file "Lorg" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN               ! Clear the alpha screen
110   GINIT                      ! Initialize various graphics parameters
120   PLOTTER IS CRT,"INTERNAL"  ! Use the internal screen
130   GRAPHICS ON                ! Turn on the graphics screen
140   SHOW 0,10,10.5,0           ! Isotropic scaling: Left/Right/Bottom/Top
150   FRAME                      ! Draw a box around the screen
160   FOR Lorg=1 TO 9            ! Loop on LORG parameters
170     LORG 2                   ! Left-center origin for the "LORG n ="
180     CSIZE 4                  ! Characters cell 4 GDUs high
190     MOVE 0,Lorg              ! Move to position for "LORG n =" label
200     LABEL "LORG";Lorg;"="    ! Write the label
210     MOVE 8+.1,Lorg+.1        ! \
220     DRAW 8-.1,Lorg-.1        !  \
230     MOVE 8-.1,Lorg+.1        !   >  Draw an "X" to show where pen is
240     DRAW 8+.1,Lorg-.1        !  /
250     LORG Lorg                ! Specify LORG for "TEST",
260     CSIZE 6                  ! ...and larger letters
270     MOVE 8,Lorg              ! Move the center of the "X"
280     LABEL "TEST"             ! Write "TEST", using current LORG
290   NEXT Lorg                  ! And so forth
300   END
```

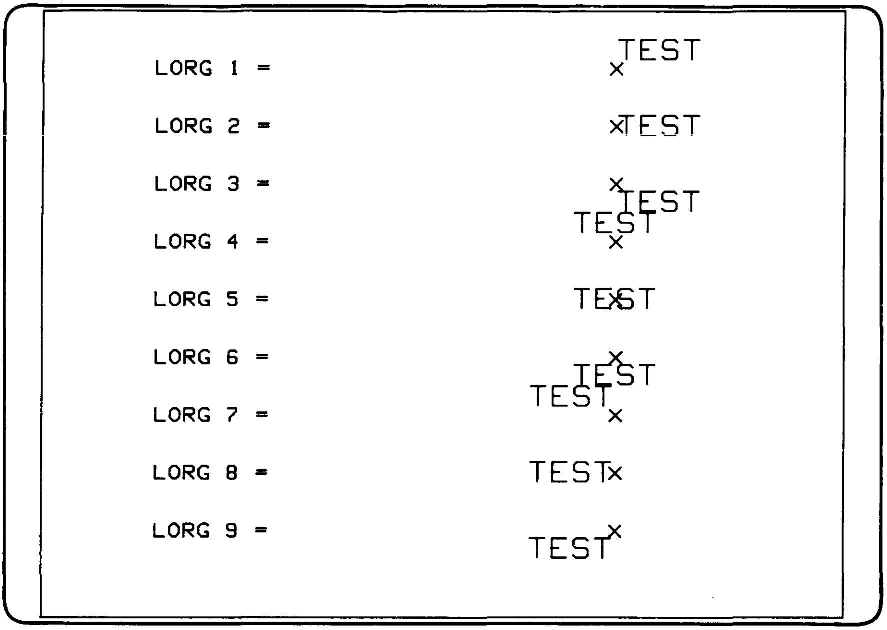


Figure 2-4. Label Origins (Lorg)

The ×'s indicate where the pen was moved to before labelling the word "TEST". What this diagram means is that, for example, if LORG 1 is in effect, and you move to 4,5 to write a label, the lower left of that label would be at 4,5. This automatically compensates for the character size, aspect ratio, and label length. It makes no difference whether there is an odd or even number of characters in the label. If LORG 6 had been in effect, and you had moved to 4,5, the center of the top edge of the label would be at 4,5. You can readily see how useful this statement is in centering labels, both horizontally and vertically.

The third statement you need to know is LDIR, meaning label direction. This specifies the angle at which the subsequent labels will be drawn. The angle is specified in the current angular units, and is either DEG (degrees) or RAD (radians). For example, assuming degrees is the current angular mode:

`LDIR 0`	Writes label horizontally to the right.
`LDIR 90`	Writes label vertically, ascending.
`LDIR 14`	Writes label ascending a gentle slope, up and right.
`LDIR 180`	Writes label upside down.
`LDIR 270`	Writes label vertically, descending.

In the program below, (which is in the file "Ldir" on the *Manual Examples Disc*) you'll note that LORG 2 was specified, and this remained in effect for many LDIRs. Each label is centered on the left edge (relative to the *label*, remember).

```
100   CLEAR SCREEN                  ! Clear the alpha display
110   GINIT                         ! Initialize various graphics parameters
120   PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
130   GRAPHICS ON                   ! Turn on the graphics screen
140                                 ! (Series 200 computers)
150   FRAME                         ! Draw a box around the screen
160   WINDOW -1,1,-1.1,1            ! Anisotropic units; Left/Right/Bottom/Top
170   DEG                           ! Angular mode: Degrees
180   LORG 2                        ! Label origin is left center
190   FOR Angle=0 TO 350 STEP 10    ! Every 10 degrees
200     LDIR Angle                  ! Labelling angle
210     MOVE 0,0                    ! Move to center of screen
220     LABEL "-------LDIR";Angle   ! Write using the current LDIR
230   NEXT Angle                    ! And so on
240   END
```

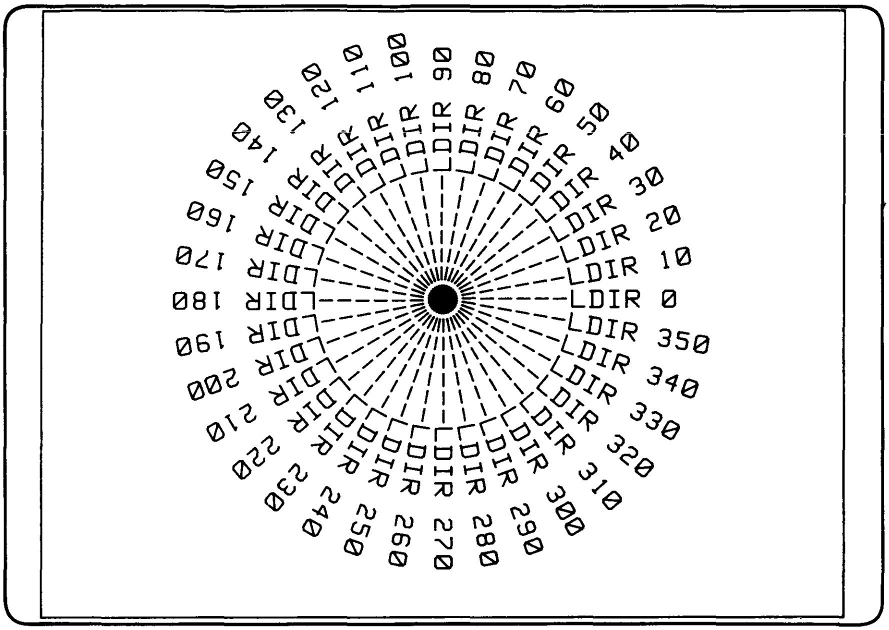


Figure 2-5. Changing Label Directions (Ldir)

The label origin specified by LORG is relative to the *label*, not the plotting surface, and it is independent of the current label direction. For example, if you have specified

```
LORG 3
DEG
LDIR 90
MOVE 6,8
```

and then write the label, it is written going straight up, not horizontally. Therefore, it is the upper left corner of the label which is at point 6,8 *relative to the rotated label.* Relative to the plotting device, however, it is the lower left corner of the label which is at 6,8 (in this example) because the label has been rotated.

Now we can discuss the statement which actually causes labels to be written: LABEL. LABEL takes into account the most recently-specified CSIZE, LDIR and LORG when it writes a label. You must position the label, however, by using (for example) a MOVE statement to get to the point at which you want the label to be placed.

All four statements have been utilized in the following update to our progressive plotting program. You may load this program from file "SinLabel" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                     ! Clear the alpha display
110   GINIT                            ! Initialize various graphics parameters.
120   PLOTTER IS CRT,"INTERNAL"        ! Use the internal screen
130   GRAPHICS ON                      ! Turn on the graphics screen
140   X_gdu_max=100*MAX(1,RATIO)       ! Determine how many GDUs wide the screen is
150   Y_gdu_max=100*MAX(1,1/RATIO)     ! Determine how many GDUs high the screen is
160   LORG 6                           ! Reference point: center of top of label
170   MOVE X_gdu_max/2,Y_gdu_max       ! Move to middle of top of screen
180   LABEL "VOLTAGE VARIANCE"         ! Write title of plot
190   DEG                              ! Angular mode is degrees (used in LDIR)
200   LDIR 90                          ! Specify vertical labels
210   CSIZE 3.5                        ! Specify smaller characters
220   MOVE 0,Y_gdu_max/2               ! Move to center of left edge of screen
230   LABEL "Voltage"                  ! Write Y-axis label
240   LORG 4                           ! Reference point: center of bottom of label
250   LDIR 0                           ! Horizontal labels again
260   MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
270   LABEL "Time (seconds)"           ! Write X-axis label
280   VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
! Define subset of screen area
290   FRAME                            ! Draw a box around defined subset
300   WINDOW 0,100,16,18               ! Anisotropic scaling: left/right/bottom/top
310   FOR X=2 TO 100 STEP 2            ! Points to be plotted...
320     PLOT X,RND+16.5                ! Get a data point and plot it against X
330   NEXT X                           ! et cetera
340   END
```

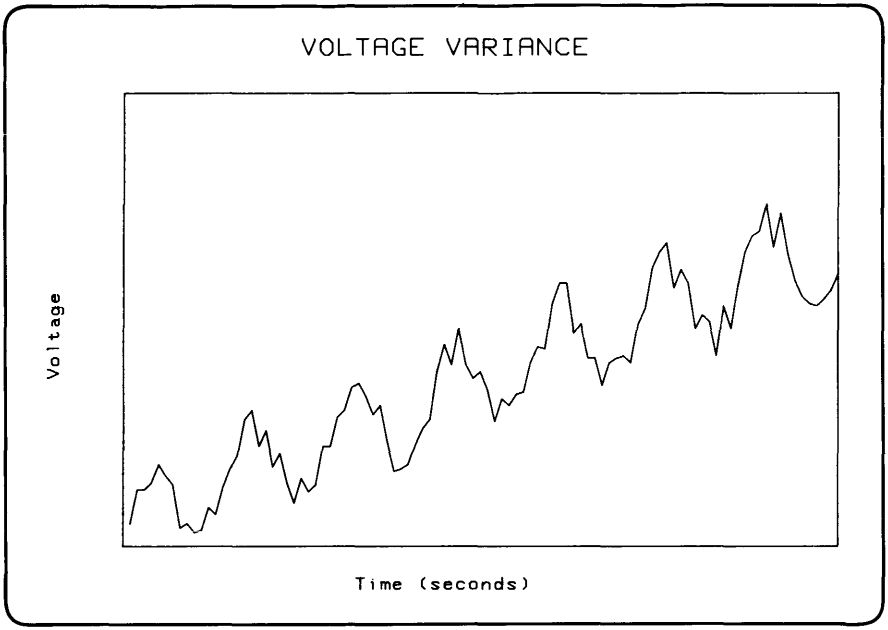


Figure 2-6. Example of Labeling (SinLabel)

Many times it's nice to have the most important titles not only in large letters, but **bold** letters, to make them stand out even more. It is possible to achieve this effect by plotting the label several times, moving the label origin just slightly each time. In the following version of the program (in file "SinLabel2" on your *Manual Examples Disc*), notice lines 180 through 210. The loop variable, I, goes from −.3 to .3 by tenths. This is the offset in the X direction (in GDUs[1]) of the label origin. Since this is being labelled with LORG 6 in effect, the label origin (the point moved to immediately prior to labelling) represents the center of the top edge of the label.

[1] Technically, a MOVE uses UDUs for its units, but until a SHOW or WINDOW is executed, UDUs are identical to GDUs.

```
100   CLEAR SCREEN                    ! Clear the alpha display
110   GINIT                           ! Initialize various graphics parameters.
120   PLOTTER IS CRT,"INTERNAL"       ! Use the internal screen
130   GRAPHICS ON                     ! Turn on the graphics screen
140   X_gdu_max=100*MAX(1,RATIO)      ! Determine how many GDUs wide the screen is
150   Y_gdu_max=100*MAX(1,1/RATIO)    ! Determine how many GDUs high the screen is
160   LORG 6                          ! Reference point: center of top of label
170   FOR I=-.3 TO .3 STEP .1         ! Offset of X from starting point
180     MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
190     LABEL "VOLTAGE VARIANCE"      ! Write title of plot
200   NEXT I                          ! Next position for title
210   DEG                             ! Angular mode is degrees (used in LDIR)
220   LDIR 90                         ! Specify vertical labels
230   CSIZE 3.5                       ! Specify smaller characters
240   MOVE 0,Y_gdu_max/2              ! Move to center of left edge of screen
250   LABEL "Voltage"                 ! Write Y-axis label
260   LORG 4                          ! Reference point: center of bottom of label
270   LDIR 0                          ! Horizontal labels again
280   MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
290   LABEL "Time (seconds)"          ! Write X-axis label
300   VIEWPORT .1*X_gdu_max,.99*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
! Define subset of screen area
310   FRAME                           ! Draw a box around defined subset
320   WINDOW 0,100,16,18              ! Anisotropic scaling: left/right/bottom/top
330   FOR X=2 TO 100 STEP 2           ! Points to be plotted...
340     PLOT X,RND+16.5               ! Get a data point and plot it against X
350   NEXT X                          ! et cetera
360   END
```

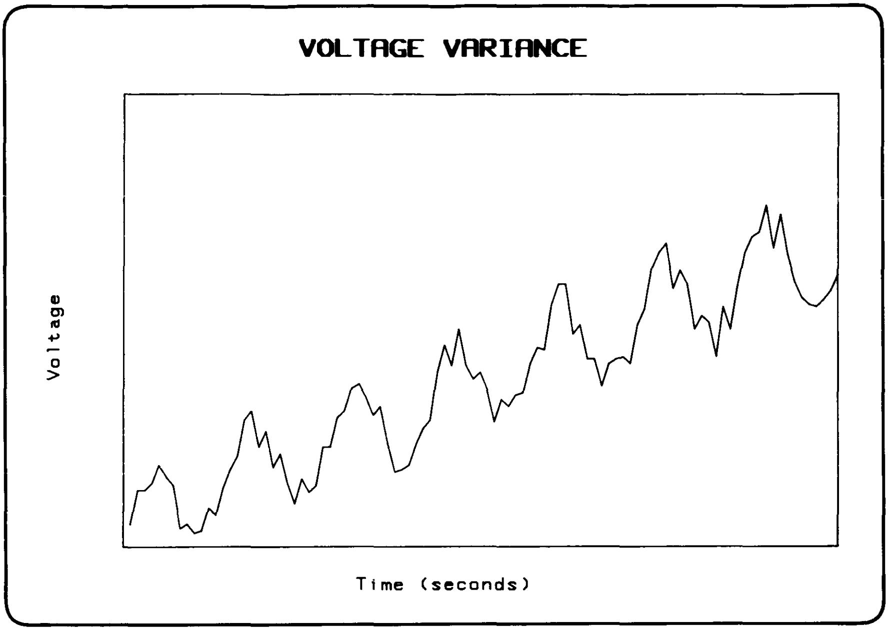


Figure 2-7. Bold Labels (SinLabel2)

This method can also be used for offsetting in the Y direction. Or, offset both X and Y. This will give you characters which are thick in a diagonal direction, which makes them look like they are coming out of the page at you. However, a more typical bolding is produced by offsetting only in the X direction.

Now we know what we are measuring — voltage vs. time — but still the units are not shown. As we saw in the last chapter, what is needed is an X-axis and a Y-axis, and they need to be labelled with numbers in appropriate places.

Axes and Grids

The AXES statement and the GRID statement do similar operations. We saw in the last chapter how to use the AXES statement. The GRID statement causes the major tick marks to extend all the way across the plotting surface.

Once we have the axes drawn, we must label various points along them with numbers designating the values at those points. Once again, we use the LABEL statement. You may load this program from file "SinAxes" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                  ! Clear the alpha display
110   GINIT                         ! Initialize various graphics parameters.
120   PLOTTER IS CRT,"INTERNAL"     ! Use the internal screen
130   GRAPHICS ON                   ! Turn on the graphics screen
140   X_gdu_max=100*MAX(1,RATIO)    ! Determine how many GDUs wide the screen is
150   Y_gdu_max=100*MAX(1,1/RATIO)  ! Determine how many GDUs high the screen is
160   LORG 6                        ! Reference point: center of top of label
170   FOR I=-.3 TO .3 STEP .1       ! Offset of X from starting point
180     MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
190     LABEL "VOLTAGE VARIANCE"    ! Write title of plot
200   NEXT I                        ! Next position for title
210   DEG                           ! Angular mode is degrees (used in LDIR)
220   LDIR 90                       ! Specify vertical labels
230   CSIZE 3.5                     ! Specify smaller characters
240   MOVE 0,Y_gdu_max/2            ! Move to center of left edge of screen
250   LABEL "Voltage"               ! Write Y-axis label
260   LORG 4                        ! Reference point: center of bottom of label
270   LDIR 0                        ! Horizontal labels again
280   MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
290   LABEL "Time (seconds)"        ! Write X-axis label
300   VIEWPORT .1*X_gdu_max,.98*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
! Define subset of screen area
310   FRAME                         ! Draw a box around defined subset
320   WINDOW 0,100,16,18            ! Anisotropic scaling: left/right/bottom/top
330   AXES 1,.05,0,16,10,5,3        ! Draw axes with appropriate ticks
340   CLIP OFF                      ! So labels can be outside VIEWPORT limits
350   CSIZE 2.5,.5                  ! Smaller chars for axis labelling
360   LORG 6                        ! Ref. pt: Top center     |\
```

```
370   FOR I=0 TO 100 STEP 10        ! Every 10 units         | \
380     MOVE I,15.99                ! A smidgeon below X-axis |  >  Label X-axis
390     LABEL USING "#,K";I         ! Compact; no CR/LF      | /
400   NEXT I                        ! et sequens             |/
410   LORG 8                        ! Ref. pt: Right center  |\
420   FOR I=16 TO 18 STEP .25       ! Every quarter          | \
430     MOVE -.5,I                  ! Smidgeon left of Y-axis |  > Label Y-axis
440     LABEL USING "#,DD.DD";I     ! DD.DD; no CR/LF        | /
450   NEXT I                        ! et sequens             |/
460   PENUP
470   FOR X=2 TO 100 STEP 2         ! Points to be plotted
480     PLOT X,RND+16.5             ! Plot a data point
490   NEXT X
500   END
```

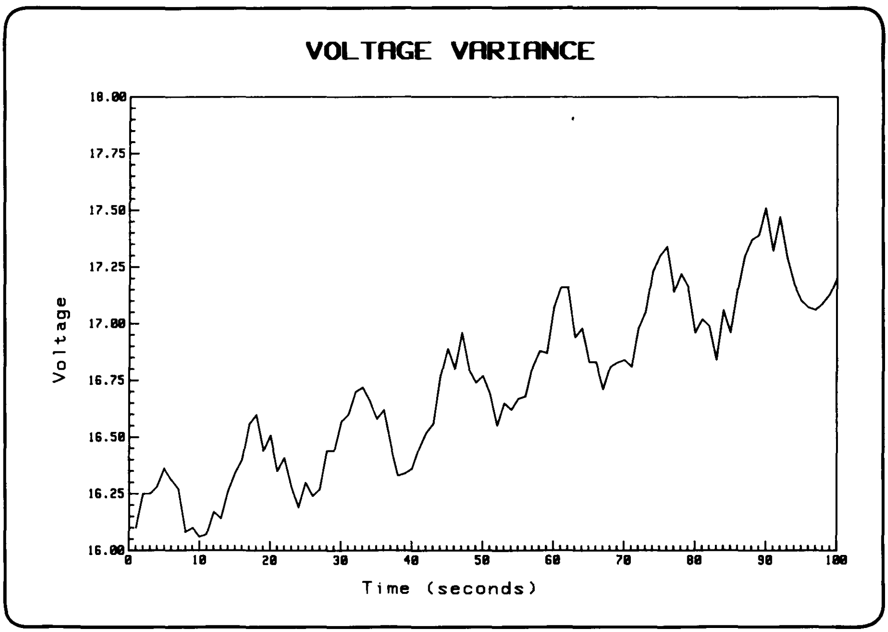


Figure 2-8. Labeled Axes (SinAxes)

Note that the tick marks drawn by the AXES statement extend only toward the interior of the graph. This was deliberate. Clipping (automatically put into effect by the VIEWPORT statement) was still active at the soft clip limits. If the CLIP OFF statement had been placed *before* the AXES statement, the tick marks would have extended on both sides of the axes. However, the axes themselves would have extended across the entire width of the hard clip limits and right through the axes' labels.

The CLIP OFF statement was necessary, though. The LABEL statement draws the letters as a series of vectors (lines), and any lines which are outside the current soft clip limits (when CLIP is ON) are cut off. Which means that had line 350 (the CLIP OFF) been missing from the program, none of the axis labels would have been drawn, since they are *all* outside the VIEWPORT area. Of course, the main titles ("VOLTAGE VARIANCE", "Voltage", and "Time (seconds)") would still have been drawn, because they are done *before* the VIEWPORT is executed.

If your graph needs to be read with more precision than the AXES statement affords, you can use the GRID statement. This is similar to AXES, except the major ticks extend across the entire soft clip area, and the minor ticks for X and Y intersect in little crosses between the grid lines. The previous program has only one change: the AXES statement has been replaced by a GRID statement.

```
GRID 5,.25,0,16,2,2,1          ! Draw grid with appropriate ticks
```

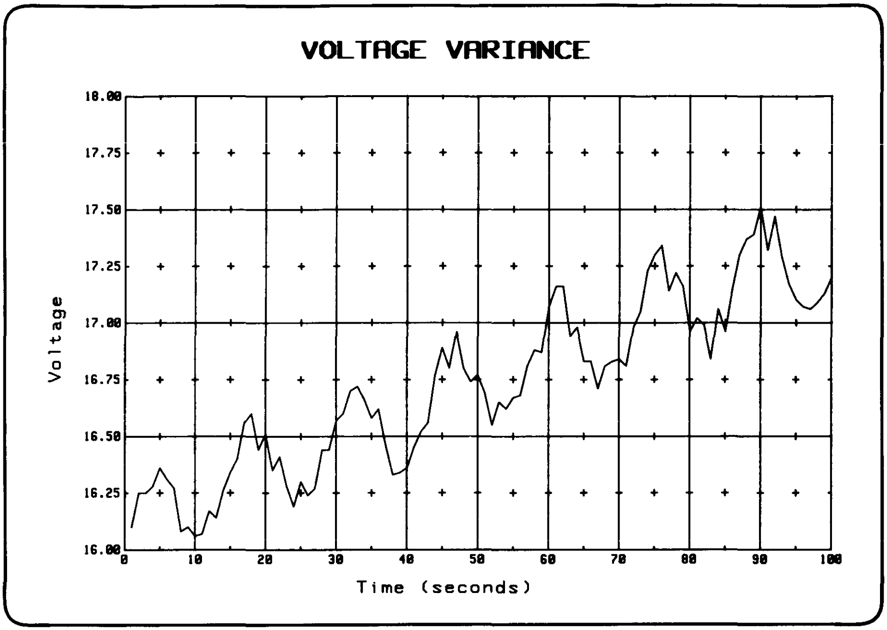


Figure 2-9. Labeled Grid

Note that not only was the keyword AXES replaced by GRID, some of the parameters were changed also. The reason for this is that the minor ticks specified in the AXES statement were so close together that the minor tick crosses drawn by the GRID statement would have overlapped. The end result would have been a grid with even the minor ticks extending all the way across the soft clip area.

Strategy: Axes vs. Grids

On many occasions, an application is defined such that there is no question as to which statement to use. Other times, however, it is not such a cut-and-dried situation and you want to weigh the alternatives carefully before setting your program in concrete. To aid you in the decision, here are some pros and cons to both statements.

Advantages of AXES:

- It executes much faster than GRID. This is for two reasons. First, there is much less calculating the computer must do, and second, there is much less actual drawing of lines the computer must do. This becomes especially evident when sending a plot to a hard-copy plotting device where physical pen must be hauled around.
- It does not clutter the plot as much. Reference points are available at the axes, but there is no question about where the data curve is. When using GRID, it is possible to lose the data curve among the reference lines if it is close to being horizontal or vertical.

Advantages of GRID:

- Interpolation and estimation are much more accurate due to the great number of reference ticks and lines; one need not estimate horizontal and vertical lines to refer back to the axis labels.
- Usually there is no need to use a FRAME statement to completely enclose the soft clip limits, as is often desired, because the major tick marks from the GRID statement would probably redraw the lines. Of course, this is dependent upon the Major Tick count.

There is a way to get the best of both worlds, however. If you want to be able to estimate data points very accurately from the finished plot, but also want to prevent the plot from appearing too cluttered, it can be done. Below is a plot drawn by a program identical to the previous one except for the GRID statement. The GRID statement used specifies exactly the same parameters as the AXES statement (two programs ago) with one exception: the Major Tick Length parameter is reduced. This causes the tick crosses (the little plus signs) to be reduced to dots. Using this strategy allows easy interpolation of data points (to the same accuracy previously used in the AXES statement), but does not clutter the graph nearly as much as normal ticks would. In fact, had we used the default minor tick length, the length of the lines making up the tick crosses would have been greater than the distance between the ticks. Thus they would have merged together to make solid lines, extending all the way across and cluttering the graph.

```
GRID 1,.05,0,16,10,5,.0001     ! Draw grid with appropriate ticks
```

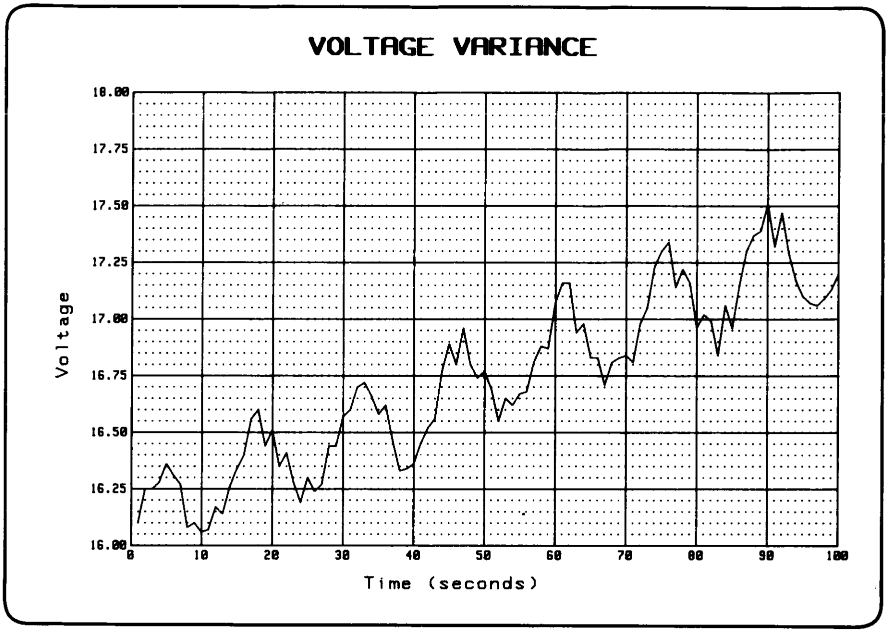


Figure 2-10. Labeled Grid with More Tick Marks

Be aware when using this strategy of making huge numbers of degenerate tick crosses that the computer still thinks of them as crosses, which means that both the horizontal and vertical components must be drawn. This looks to you like drawing and then redrawing each dot. Therefore, when sending this type of grid to a hard-copy plotter, do not be averse to starting your plot, and then reading the remainder of this chapter.

Another way to reach a compromise between ease of interpolation and lack of clutter is to use *both* AXES and GRID in the same program. Note the program below. GRID is used for the major tick lines, but since the minor tick crosses are not desired within each rectangle between the major tick lines, AXES is used to specify minor ticks. This program is in the file "SinGrdAxes" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                    ! Clear the alpha display
110   GINIT                           ! Initialize various graphics parameters.
120   PLOTTER IS CRT,"INTERNAL"       ! Use the internal screen
130   GRAPHICS ON                     ! Turn on the graphics screen
140   LORG 6                          ! Reference point: center of top of label
150   X_gdu_max=100*MAX(1,RATIO)      ! Determine how many GDUs wide the screen is
160   Y_gdu_max=100*MAX(1,1/RATIO)    ! Determine how many GDUs high the screen is
170   FOR I=-.3 TO .3 STEP .1         ! Offset of X from starting point
180     MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
190     LABEL "VOLTAGE VARIANCE"      ! Write title of plot
200   NEXT I                          ! Next position for title
210   DEG                             ! Angular mode is degrees (used in LDIR)
220   LDIR 90                         ! Specify vertical labels
230   CSIZE 3.5                       ! Specify smaller characters
240   MOVE 0,Y_gdu_max/2              ! Move to center of left edge of screen
250   LABEL "Voltage"                 ! Write Y-axis label
260   LORG 4                          ! Reference point: center of bottom of label
270   LDIR 0                          ! Horizontal labels again
280   MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
290   LABEL "Time (seconds)"          ! Write X-axis label
300   VIEWPORT .1*X_gdu_max,.98*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
! Define subset of screen area
310   WINDOW 0,100,16,18              ! Anisotropic scaling: left/right/bottom/top
320   AXES 1,.05,0,16,5,5,3           ! Draw axes intersecting at lower left
330   AXES 1,.05,100,18,5,5,3         ! Draw axes intersecting at upper right
340   GRID 10,.25,0,16,1,1            ! Draw grid with no minor ticks
350   CLIP OFF                        ! So labels can be outside VIEWPORT limits
360   CSIZE 2.5,.5                    ! Smaller chars for axis labelling
370   LORG 6                          ! Ref. pt: Top center      |\
380   FOR I=0 TO 100 STEP 10          ! Every 10 units           | \
390     MOVE I,15.99                  ! A smidgeon below X-axis  |  >  Label X-axis
400     LABEL USING "#,K";I           ! Compact; no CR/LF        | /
410   NEXT I                          ! et sequens               |/
420   LORG 8                          ! Ref. pt: Right center     |\
430   FOR I=16 TO 18 STEP .25         ! Every quarter             | \
440     MOVE -.5,I                    ! Smidgeon left of Y-axis   |  > Label Y-axis
450     LABEL USING "#,DD.DD";I       ! DD.D; no CR/LF            | /
460   NEXT I                          ! et sequens                |/
470   PENUP                           ! LABEL statement leaves the pen down
480   FOR X=2 TO 100 STEP 2           ! Points to be plotted...
490     PLOT X,RND+16.5               ! Get a data point and plot it against X
500   NEXT X                          ! et cetera
510   END
```

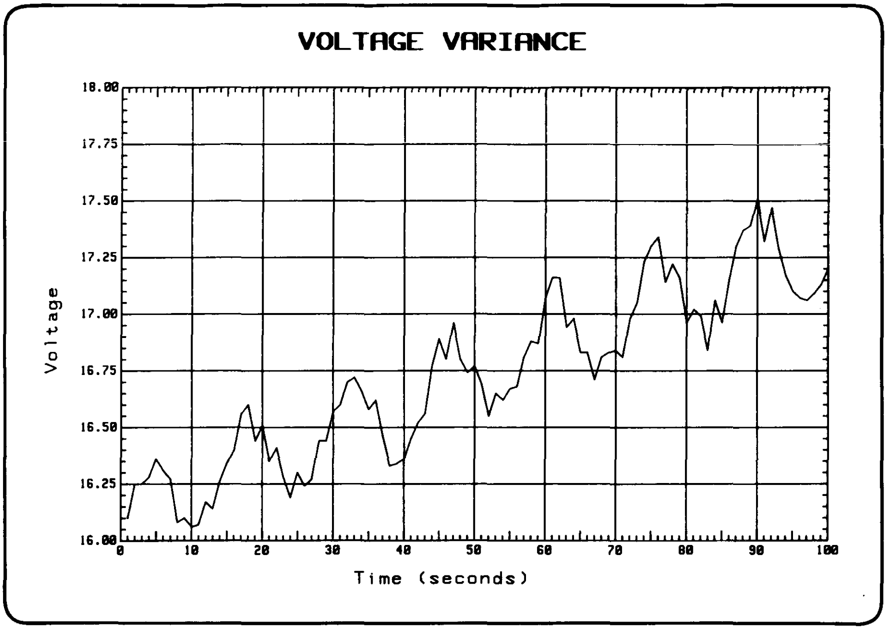


Figure 2-11. Using AXES and GRID (SinGrdAxes)

Note that *two* AXES statements were used. The parameters are identical save for the position of the intersection. The first AXES specifies an intersection position of 0,16: the lower left corner of the soft clip area. The second specifies an intersection position of 100,18: the upper right corner of the soft clip area.

Also note that the FRAME statement was removed; the lines around the soft clip limit were being drawn by both the pair of AXES statements and the GRID statement anyway.

This is the final version of our illustrative series of examples. The series of examples was used to help you grow in ability to create graphics programs and see how they can be structured to illustrate information generated from raw data (hypothetically input using the RND function). In actual practice the data source could have been a voltmeter or other device.

Miscellaneous Graphics Concepts

Clipping

Something that occurs completely "behind the scenes" in your computer when drawing is a process called **clipping**. Clipping is the process whereby lines that extend over the defined edges of the drawing surface are cut off at those edges. There are two different clipping boundaries at all times: the **soft clip limits** and the **hard clip limits**. The hard clip limits are the absolute boundaries of the plotting surface, and under no circumstances can the pen go outside of these limits. The soft clip limits are user-definable limits, and are defined by the CLIP statement.

```
CLIP 10,20.5,Ymin,Ymax
```

This statement defines the soft clip boundaries only; hard clip limits are completely unaffected. After this statement has been executed, all lines which attempt to go outside the X limits (in UDUs) of 10 and 20.5, or the Y limits (in UDUs) of `Ymin` and `Ymax` will be truncated at the appropriate edge. Clipping *at the soft clip limits* can be turned off by the statement:

```
CLIP OFF
```

and it can be turned back on, using the same limits, by

```
CLIP ON
```

If you want the soft clip limits to be somewhere else, use the CLIP statement with four different limits. Only one set of soft clip limits can be in effect at any one time. Clipping at the hard clip limits cannot be disabled.

The VIEWPORT statement, in addition to defining how WINDOW coordinates map into the VIEWPORT area, turns on clipping at the specified VIEWPORT edges.

Drawing Modes

On a monochromatic CRT, there are three different drawing modes available. (For selecting pens with a color CRT, see the "Color Graphics" chapter.) The three pens perform the following actions:

Table 2-1. Monochromatic Pens

Pen Number	Function
1	Draws lines (turns on pixels)
−1	Erases lines (turns off pixels)
0	Complements lines (changes pixels' states)

A characteristic of drawing with pen −1 or pen 1 is that if a line crosses a previously-drawn line, the intersection will be the same "color" as the lines themselves. When drawing with pen 0, and a line crosses a previously-drawn line, the intersection becomes the opposite state of the lines. For example, assume a black background (like right after a GCLEAR). You select PEN 0, then draw a pair of AXES. When the first axis is drawn, all pixels are off, so the line being drawn causes all pixels to be turned on along its length. However, when the second axis is drawn, it will turn on pixels until it gets to the other axis. At that point, the pixel is on, so it gets turned *off*. After that, the rest of the pixels are off, so they are again turned on.

This concept is illustrated by the following program (file "Pen" on the *Manual Examples Disc*). The listing is given so you can see it in action, but since it is a dynamic display, it makes little sense to show a snapshot of it. Line 150 of the program defines the type of operation the program will exhibit. If `Pen` equals zero, all lines will complement, because lines 610 and 680 select pen −0 and +0, which are identical. When you wish to change the program to drawing and erasing mode, change line 150 to say `Pen=1`. Then lines 610 and 680 will select pens −1 and +1, respectively.

```
100   CLEAR SCREEN                                 ! Clear the alpha display
110   INTEGER Polygon,Polygons,Side,Sides,Pen      ! Make loops faster
120   Polygons=20                                  ! How many polygons?
130   Sides=3                                      ! How many sides apiece?
140   Pen=0                                        ! 1: Draw/erase; 0: Complement
150   ALLOCATE INTEGER X(0:Polygons-1,1:Sides),Y(0:Polygons-1,1:Sides)
160   ALLOCATE INTEGER Dx(Sides),Dy(Sides)
170   RANDOMIZE                                    ! Different each time
180   GINIT                                        ! Initialize graphics parameters
190   PLOTTER IS CRT,"INTERNAL"                    ! Use the internal screen
200   GRAPHICS ON                                  ! Turn on graphics screen
210   WINDOW 0,511,0,389                           ! Integer arithmetic is faster
220   PEN Pen                                      ! Select appropriate pen
230   FOR Side=1 TO Sides                          ! For each vertex...
240     X(0,Side)=RND*512                          ! ...define a starting point...
250     Y(0,Side)=RND*390                          ! ...for both X and Y...
260     PLOT X(0,Side),Y(0,Side)                   ! ...then draw to that point.
270   NEXT Side                                    ! et cetera
280   IF Sides>2 THEN PLOT X(0,1),Y(0,1)           ! If simple line, don't close
290   GOSUB Define_deltas                          ! Get dx and dy for each vertex
300   FOR Polygon=1 TO Polygons-1                  ! Draw all the polygons
310     PENUP                                      ! Don't connect polygons
320     FOR Side=1 TO Sides                        ! Each vertex of each polygon
330       Temp=X(Polygon-1,Side)+Dx(Side)          ! Avoid recalculation
340       IF Temp>511 THEN                         ! \
350         Dx(Side)=-Dx(Side)                     !  \
360       ELSE  ! (it's not off right side)        !   > Is X out of range?
370         IF Temp<0 THEN Dx(Side)=-Dx(Side)      !  /
380       END IF  ! (off right side?)              ! /
390       X(Polygon,Side)=X(Polygon-1,Side)+Dx(Side)  ! Calculate next X
400       Temp=Y(Polygon-1,Side)+Dy(Side)          ! Avoid recalculation
410       IF Temp>389 THEN                         ! \
420         Dy(Side)=-Dy(Side)                     !  \
430       ELSE  ! (it's not off top)               !   > Is Y out of range?
440         IF Temp<0 THEN Dy(Side)=-Dy(Side)      !  /
450       END IF  ! (off the top?)                 ! /
460       Y(Polygon,Side)=Y(Polygon-1,Side)+Dy(Side)  ! Calculate new Y
470       PLOT X(Polygon,Side),Y(Polygon,Side)     ! Draw line to new point
480     NEXT Side                                  ! Loop for next side of polygon
490     IF Sides>2 THEN PLOT X(Polygon,1),Y(Polygon,1)! If line, don't close
500   NEXT Polygon                                 ! Get each polygon
510   New=0                                        ! Start re-use at entry 0
520   ON CYCLE 10 GOSUB Define_deltas              ! Change deltas periodically
530   LOOP                                         ! Ad infinitum...
```

```
540     IF New=0 THEN                            ! Boundary condition?
550       Previous=Polygons-1                    ! Start re-using over
560     ELSE  ! (new>0)
570       Previous=(Previous+1) MOD Polygons     ! Re-use next entry
580     END IF  ! (new=0?)
590     PENUP                                    ! Don't connect polygons
600     PEN -Pen                                 ! This works either way for Pen
610     DISABLE                                  ! Don't interrupt in "Side" loop
620     FOR Side=1 TO Sides                      ! \
630       PLOT X(New,Side),Y(New,Side)           !  \
640     NEXT Side                                !   > Erase oldest line
650     IF Sides>2 THEN PLOT X(New,1),Y(New,1)   !  /
660     PENUP                                    ! /
670     PEN Pen                                  ! Drawing pen
680     FOR Side=1 TO Sides                      ! \
690       Temp=X(Previous,Side)+Dx(Side)         !  \
700       IF Temp>511 THEN                       !   \
710         Dx(Side)=-Dx(Side)                   !    \
720       ELSE                                   !     \
730         IF Temp<0 THEN Dx(Side)=-Dx(Side)    !      \
740       END IF                                 !       \
750       X(New,Side)=X(Previous,Side)+Dx(Side)  !        \
760       Temp=Y(Previous,Side)+Dy(Side)         !         \  Draw the new line
770       IF Temp>389 THEN                       !         /  same way as before.
780         Dy(Side)=-Dy(Side)                   !        /
790       ELSE                                   !       /
800         IF Temp<0 THEN Dy(Side)=-Dy(Side)    !      /
810       END IF                                 !     /
820       Y(New,Side)=Y(Previous,Side)+Dy(Side)  !    /
830       PLOT X(New,Side),Y(New,Side)           !   /
840     NEXT Side                                !  /
850     IF Sides>2 THEN PLOT X(New,1),Y(New,1)   ! /
860     ENABLE                                   ! Interrupts OK again
870     New=(New+1) MOD Polygons                 ! Next one to re-use.
880   END LOOP                                   ! End of infinite loop
890 Define_deltas:  ! ------------------------------------------------------
900   FOR Side=1 TO Sides                        ! For each vertex
910     Dx(Side)=RND*3+2                         ! Magnitude of this dx
920     IF RND<.5 THEN Dx(Side)=-Dx(Side)        ! Sign of this dx
930     Dy(Side)=RND*3+2                         ! Magnitude of this dy
940     IF RND<.5 THEN Dy(Side)=-Dy(Side)        ! Sign of this dy
950   NEXT Side                                  ! et cetera
960   RETURN                                     ! back to the main program
970   END
```

Observe when running the program in complementing mode that a pixel is on only if it has been acted upon by an odd number of line segments.

Selecting Line Types

When a graph is attempting to convey several different kinds of information, colors are often used: The red curve signifies one thing, the blue curve signifies another thing, etc. But when only one color is available, as on a monochromatic CRT, this method cannot be used. Something that *can* be used, however, is different line types. Even on a monochrome CRT, it makes sense to say that the solid line signifies one thing, the dotted line signifies another thing, and the dashed line signifies still another.

There are ten line types available:

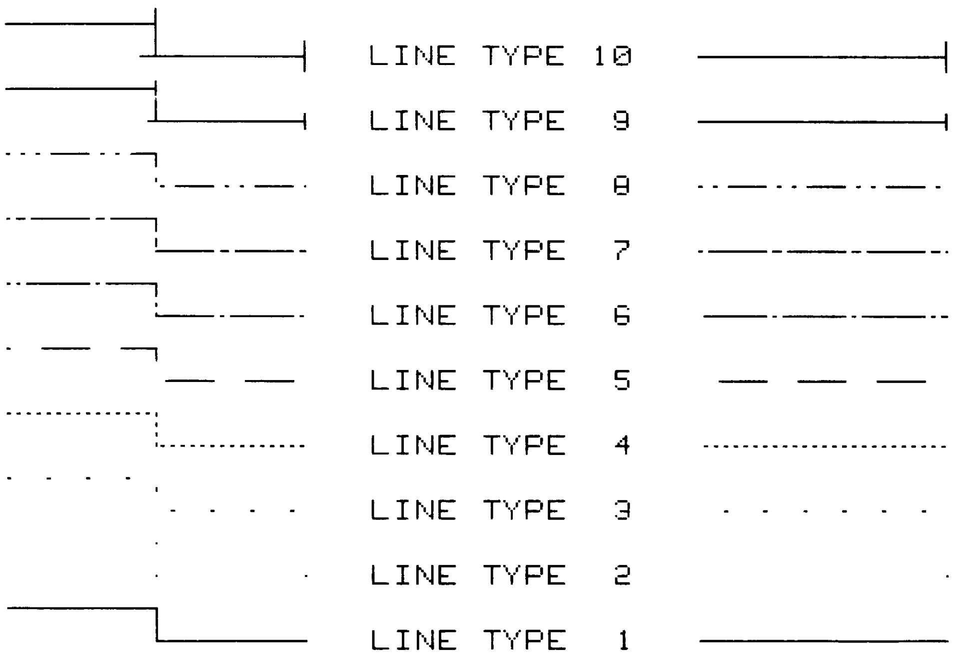


Figure 2-12. Series 200 Line Types

As you can see, LINE TYPE 1 draws a solid line. LINE TYPE 2 draws only the end points of the lines and is the same as moving to a new point, dropping the pen, lifting the pen, and repeating. LINE TYPEs 3 through 8 are patterned sequences of on and off. With these, the length of each pattern, i.e, the distance the line extends before the on/off pattern begins to repeat, can be specified by supplying a second parameter in the LINE TYPE statement. This second parameter specifies distance in GDUs. For example,

```
LINE TYPE 5,15
```

tells the computer to start using a simple dashed line, and to proceed a total of 15 GDUs before starting the pattern over. On the CRT, the repeat length will be rounded to a multiple of five, with a minimum value of five.

LINE TYPEs 9 and 10 are solid lines with a minor and major tick mark at the end of each line, respectively. The tick mark will be either horizontal or vertical. The orientation of the tick marks will be whatever is farther from the angle of the line just drawn. For example, if you draw a line at a thirty-degree angle, it is closer to being horizontal than it is to being vertical. Thus, tick mark at the end of the line will be vertical. The value for major tick size is 2 GDUs, and minor tick length is one half the major tick length.

For all line types, the computer remembers where in the pattern a line segment ended. Therefore, when you start drawing another line segment, the line pattern will continue from where it left off. If you want the pattern to start over, just re-execute the LINE TYPE statement.

Storing and Retrieving Images

If a picture on the screen takes a long time to draw, or the image is used often, it may be advisable to store the image itself—*not* the commands used to draw the image—in memory or on a file.

This may be done with the GSTORE command. First, you must have an INTEGER array of sufficient size to hold all the data in the graphics raster. The array size varies depending on what computer system you have in general and what monitor you have in particular. A formula for calculating array size is:

A monochromatic display has 1 bit per pixel. The Model 236C color computer has 4 bits per pixel. The Series 300 color monitors have either 4, 6 or 8 bits per pixel. But rather than having to get intimately involved with screen resolution and the number of bits per pixel, there is a shortcut. The fifth and sixth elements of the integer array passed back by GESCAPE operation selector 3 specify the number of rows and columns an integer array must have to contain the entire graphics image. For example:

```
      ...
20  INTEGER A(1:6)
30  GESCAPE CRT,3;A(*)
40  PRINT USING "K";"Array must have ";A(5)*A(6);" elements (",A(5),"x",
A(6),")."
50  ALLOCATE Gscreen(A(5),A(6))
60  GSTORE Gscreen(*)
      ...
```

The array `Gscreen` is allocated of the size specified by the "rows" and "columns" numbers from the GESCAPE return array. This array holds the picture itself, and it doesn't care how the information got to the screen, or in what order the different parts of the picture were produced.

In the following program, the image is drawn with normal plotting commands, and then, *after the fact*, the image is read from the graphics area in memory, and placed into the array. After the array is filled by the GSTORE, a curve is plotted on top of the image already there. Then, turning the knob changes the value of a parameter, and a different curve results. *But we do not have to replot the grid, axes and labels.* We merely need to GLOAD the image (which has everything but the curve and the current parameter value). This allows the curve to be inspected almost in real time. This program is contained in file "Gstore" on the *Manual Examples Disc.*

This progrom uses the GSTORE method mentioned above.

```
100   CLEAR SCREEN                     ! Clear the alpha display
110   GINIT                            ! Initialize various graphics parameters.
120   PLOTTER IS CRT,"INTERNAL"        ! Use the internal screen
130   INTEGER A(1:6)                   ! Declare array A
140   GESCAPE CRT,3;A(*)               ! Store operation selector 3 info. in A(*).
150   ALLOCATE INTEGER Screen(1:A(5),1:A(6))
160   GRAPHICS ON                      ! Turn on the graphics screen
170   CSIZE 6                          ! Large letters for main title
180   LORG 6                           ! Reference point: center of top of label
190   X_gdu_max=100*MAX(1,RATIO)       ! Determine how many GDUs wide the screen is
200   Y_gdu_max=100*MAX(1,1/RATIO)     ! Determine how many GDUs high the screen is
210   FOR I=-.25 TO .25 STEP .1        ! Offset of X from starting point
```

```
220     MOVE X_gdu_max/2+I,Y_gdu_max! Move to about middle of top of screen
230     LABEL "Blackbody Radiation" ! Write title of plot
240   NEXT I                        ! Next position for title
250   CSIZE 4                       ! Smaller letters for temperature legend
260   MOVE X_gdu_max/2,Y_gdu_max*.95! Right below main title
270   LABEL "Temperature (K):      "! Label offset to left so value will fit
280   DEG                           ! Angular mode is degrees (used in LDIR)
290   LDIR 90                       ! Specify vertical labels
300   CSIZE 3.5                     ! Specify smaller characters
310   MOVE 0,Y_gdu_max/2            ! Move to center of left edge of screen
320   LABEL "Intensity of Radiation"! Write Y-axis label
330   LORG 4                        ! Reference point: center of bottom of label
340   LDIR 0                        ! Horizontal labels again
350   MOVE X_gdu_max/2,.07*Y_gdu_max! X: center of screen; Y: above key labels
360   LABEL "Wavelength (microns)"  ! Write X-axis label
370   VIEWPORT .1*X_gdu_max,.98*X_gdu_max,.15*Y_gdu_max,.9*Y_gdu_max
! Define subset of screen area
380   Xmin=-4                       ! \
390   Xmax=3                        !  \
400   Xrange=Xmax-Xmin              !   \
410   Dx=.1                         !    \   Calculate X and Y internal data
420   Ymin=-5                       !    /
430   Ymax=25                       !   /
440   Yrange=Ymax-Ymin              !  /
450   Dy=1                          ! /
460   WINDOW Xmin,Xmax,Ymin,Ymax    ! Anisotropic scaling: left/right/bottom/top
470   CLIP OFF                      ! So labels can be outside VIEWPORT limits
480   FOR Decade=Xmin TO Xmax       !              |\
490     FOR Units=1 TO 1+8*(Decade<Xmax)!          | \
500       X=Decade+LGT(Units)       !              |  \
510       MOVE X,Ymin               !              |   > Draw logarithmic X-axis
520       DRAW X,Ymax               !              |  /
530     NEXT Units                  !              | /
540   NEXT Decade                   !              |/
550   FOR X=Xmin TO Xmax STEP Dx*10 !                            |\
560     LORG 6                      !                            | \
570     CSIZE 3                     !                            |  \
580     MOVE X,Ymin-Yrange*.01      ! A smidgeon below X-axis    |   \
590     LABEL USING "#,K";"10 "     ! Compact; no CR/LF          |    \  Label the
600     CSIZE 2                     !                            |    /  X-axis
610     LORG 1                      !                            |   /
620     MOVE X+Xrange*.01,Ymin-Yrange*.03  !                     |  /
630     LABEL USING "#,K";X         !                            | /
640   NEXT X                        ! et sequens                 |/
650   CLIP ON                       !  | \
```

```
660   AXES Xrange,Dy,Xmin,Ymin,1,5  !  |  \
670   AXES Xrange,Dy,Xmax,Ymax,1,5  !  |   >  Only powers of 10 on Y-axis
680   GRID 1,Dy*5,Xmin,Ymin         !  |  /
690   CLIP OFF                      !  | /
700   FOR Y=Ymin TO Ymax STEP Dy*5  ! Logarithmic Y-axis        |\
710     CSIZE 4                     ! Big chars for "10"        | \
720     LORG 8                      !                           |  \
730     MOVE Xmin-Xrange*.03,Y      ! Smidgeon left of Y-axis   |   \
740     LABEL USING "#,K";"10"      !                           |    \  Label the
750     CSIZE 2                     ! Small chars for exponent  |    /  Y-axis
760     LORG 1                      !                           |   /
770     MOVE Xmin-Xrange*.025,Y+Yrange*.01  !                   |  /
780     LABEL USING "#,K";Y         ! Compact; no CR/LF         | /
790   NEXT Y                        ! et sequens                |/
800   ! Here is where the action starts.......................................
810   GSTORE Screen(*)              ! Store the screen image in the array
820   CSIZE 4                       ! Same size letters as before
830   LORG 1                        ! Lower left label origin
840   Per=10                        ! Number of knob pulses before action taken
850   Mantissa=9                        ! \  These three statements define the
860   Exponent=2                        !  > temperature in a way which can be
870   Temperature=Mantissa*10^Exponent  ! /  changed logarithmically.
880   Rotation=10                   ! Make the subroutine notice first pass
890   GOSUB New_curve               ! Load the screen and plot the curve
900   ON KNOB .5 GOSUB New_curve        ! Look at the knob every half a second
910 Spin:   GOTO Spin               ! Looplooplooplooplooplooplooplooplooploop
920 New_curve:   ! -------------------------------------------------------------
930   Rotation=Rotation+KNOBX       ! Accumulate knob pulses
940   IF ABS(Rotation)<Per THEN RETURN   ! If not enough, return
950   GLOAD Screen(*)               ! Load grid (in effect, erase old curve)
960   Delta=SGN(Rotation)           ! Which way was knob turned?
970   IF Mantissa=3 AND Exponent=2 AND Delta<0 OR Mantissa=2 AND Exponent=14 AND
Delta>0 THEN                       ! Reached the limits
980     BEEP 100,.01                ! Let user know
990   ELSE  ! (in range)
1000    FOR I=1 TO INT(ABS(Rotation)/Per)! Allow rapid change of temperature
1010      GOSUB Delta               ! Increment/decrement logarithmically
1020    NEXT I
1030  END IF  ! (out of range?)
1040  Temperature=Mantissa*10^Exponent   ! Build temperature value
1050  Rotation=0                    ! Start knob rotation accumulation again
1060  CLIP OFF                      ! Allow label to be written outside viewport
1070  MOVE 0,25.4                   ! Go to label location
1080  LABEL USING "K";Temperature   ! Write new temperature
1090  PENUP                         ! Label leaves pen down
1100  CLIP ON                       ! Turn clipping back on
1110  FOR X=Xmin TO Xmax STEP Dx*2  ! # data points: CEIL((Xmax-Xmin)/Dx+eps)
```

```
1120   Y=FNIntensity(10^X,Temperature)  ! Calculate intensity
1130   PLOT X,LGT(Y)               ! Get a data point and plot it against X
1140  NEXT X                       ! et cetera
1150  RETURN
1160 Delta:  ! ------------------------------------------------------------------
1170  IF Mantissa=3 AND Exponeut=2 AND Delta<0 THEN RETURN  ! \ Have you reached
1180  IF Mantissa=2 AND Exponent=14 AND Delta>0 THEN RETURN ! / a boundary yet?
1190  IF Delta>0 THEN              ! Clockwise rotation
1200   IF Mantissa=9 THEN          ! Need to increment order of magnitude yet?
1210     Exponent=Exponent+1       ! Increment order of magnitude
1220     Mantissa=1                ! Start over with mantissa
1230   ELSE  ! (mantissa<9)
1240     Mantissa=Mantissa+1       ! In the middle of an order of magnitude
1250   END IF  ! (mantissa=9?)
1260  ELSE  ! (delta<0)            ! Counterclockwise rotation
1270   IF Mantissa=1 THEN          ! Need to decrement order of magnitude yet?
1280     Exponent=Exponent-1       ! Decrement order of magnitude
1290     Mantissa=9                ! Start mantissa over again at top end
1300   ELSE  ! (mantissa>1)
1310     Mantissa=Mantissa-1       ! In the middle of an order of magnitude
1320   END IF  ! (mantissa=1?)
1330  END IF  ! (delta>0?)
1340  RETURN
1350  END
1360  ! **********************************************************************
1370 Intensity:DEF FNIntensity(Wavelength,Temperature)
1380   Intensity=37410/Wavelength^5/(EXP(14.39/(Wavelength*Temperature))-1)
1390   RETURN Intensity
1400  FNEND
```

The curve looks like the following display.

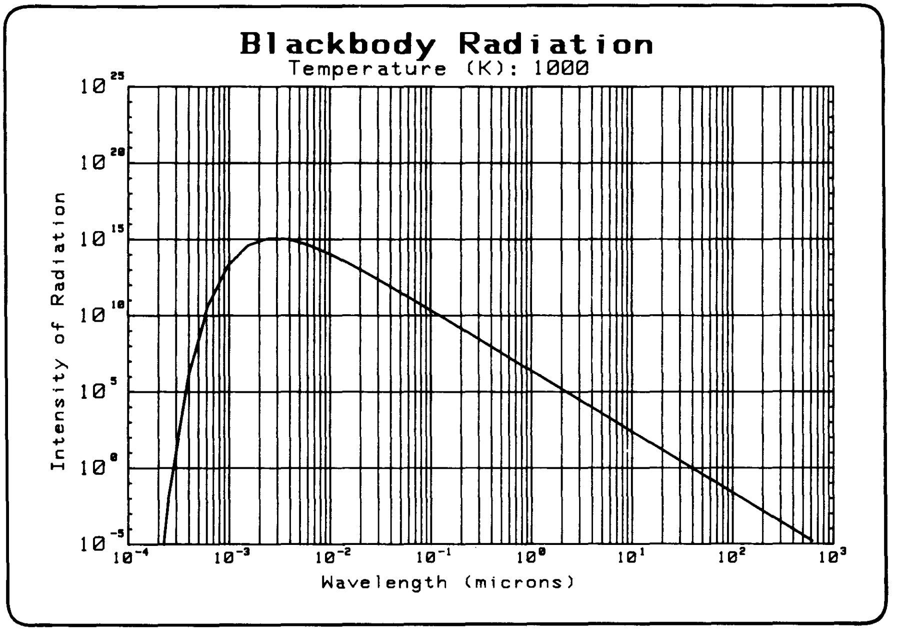


Figure 2-13. Using GSTORE and GLOAD (Gstore)

Data-Driven Plotting

Often, when plotting data points, they do not form a continuous line like the those in the last chapter's programs. One must have the ability to control the pen's position. In the last chapter, a passing reference was made to a third parameter in the PLOT statement. This third parameter is the pen-control parameter, and its function is to raise or lower the pen so many lines can be drawn with one set of data, not just one continuous line.

When using a single X-position and Y-position in a PLOT statement (as opposed to plotting an entire array; we'll cover this a little later), the third parameter is defined in the following manner. Though it need not be of type INTEGER, its value should be an integer. If it is not, it will be rounded. The third parameter is either positive or negative, and at the same time, either even or odd. The evenness/oddness of the number determines *which* action will be performed on the pen, and the sign of the number determines *when* that action will be performed: before or after the pen is moved.

Table 2-2. Pen Control Parameters

	Even (Up)	**Odd (Down)**
Positive (After)	Pen Up After Move	Pen Down After Move
Negative (Before)	Pen Up Before Move	Pen Down Before Move

The default parameter is +1—positive odd—therefore, the pen will drop after moving, and if the pen is already down, it will remain down, drawing a line. Indeed, this is what happened in the first example in Chapter 1. (Zero is considered positive.)

Following is a program (program "Lem1" on the *Manual Examples Disc*) which uses pen control. It draws a LEM (Lunar Excursion Module). In particular, see how the PLOT statement was used with an array specifier. Notice that the X and Y values are in the same array as the pen-control parameters.

```
100   CLEAR SCREEN                        ! Clear the alpha display
110   OPTION BASE 1                       ! Arrays start at one
120   DIM Lem(33,3)                       ! Data and pen-control array
130   READ Lem(*)                         ! Define the LEM data
140   GINIT                               ! Initialize various graphics parameters
150   PLOTTER IS CRT,"INTERNAL"           ! Use the internal screen
160   SHOW -10,10,-10,10                  ! Isotropic scaling
170   GRAPHICS ON                         ! Turn on the graphics screen
180   AREA INTENSITY .125,.125,.125       ! 12.5% gray
190   PLOT Lem(*)                         ! Plot the data
200 Lem:!    X    Y  Pen |  X    Y  Pen |  X    Y  Pen |  X    Y  Pen
210   DATA  0,    0, 11                   ! Start of polygon with FILL and EDGE
220   DATA  1.5,  1, -2,  2.5,  2, -1,  2.5,  3, -1,  1.5,  4, -1  ! Octagon
230   DATA -1.5,  4, -1, -2.5,  3, -1, -2.5,  2, -1, -1.5,  1, -1
240   DATA  0,    0,  7                   ! End of first polygon
250   DATA  0,    0,  6                   ! Start of polygon with FILL
260   DATA -2.5,  1, -2,  2.5,  1, -1,  2.5, -2, -1, -2.5, -2, -1  ! Box
270   DATA -2.5,  1, -1
280   DATA  0,    0,  7                   ! End of second polygon
290   DATA -2.5, -2, -2, -4.5, -4, -1, -2.5,  0, -1, -5,   -4, -2  ! Left Leg
300   DATA -4,   -4, -1
310   DATA  2.5, -2, -2,  4.5, -4, -1,  2.5,  0, -1,  5,   -4, -2  ! Rt. leg
320   DATA  4,   -4, -1
330   DATA  0,    0, 10                   ! Start of polygon with EDGE
340   DATA -0.5, -2, -2, -1,   -3, -1,  1,   -3, -1,  0.5, -2, -1  ! Nozzle
350   DATA  0,    0,  7                   ! End of third polygon
360   END
```

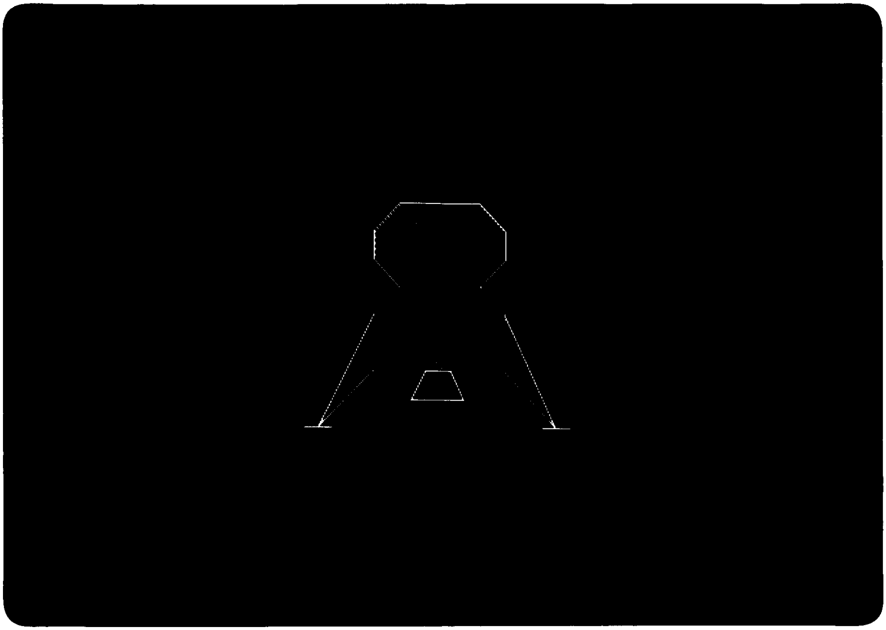


Figure 2-14. Example of Data-Driven Plotting (Lem2)

Having the pen-control parameter in a third column of the data array is generally a good strategy; it reduces the number of array names you must declare, and when you have the data points for the picture, you also have the information necessary to draw it. Nevertheless, an array must be entirely of one type, and usually you'll want the data to be REAL. If you're pressed for memory, INTEGER numbers take only one-fourth the memory REAL numbers take to store.

The PLOT keyword can plot an entire array in one statement, but you must have just one array holding both the data and pen-control parameters. That is, you cannot have the data in a two-column REAL array and the pen-control parameters in a one-column INTEGER array, unless you are plotting one point at a time, as above. The array it plots must be a single two-column or three-column array. If it is a two-column array, the pen-control parameter is assumed to be +1 for every point (pen down after move). If you have a third column in the array, the array columns are interpreted in these ways:

Table 2-3. Pen Control when Plotting Entire Arrays

Column 1	Column 2	Operation Selector	Meaning
X	Y	−2	Pen up before moving
X	Y	−1	Pen down before moving
X	Y	0	Pen up after moving (Same as +2)
X	Y	1	Pen down after moving
X	Y	2	Pen up after moving
pen number	ignored	3	Select pen
line type	repeat value	4	Select line type
color	ignored	5	Color value
ignored	ignored	6	Start polygon mode with FILL
ignored	ignored	7	End polygon mode
ignored	ignored	8	End of data for array
ignored	ignored	9	NOP (no operation)
ignored	ignored	10	Start polygon mode with EDGE
ignored	ignored	11	Start polygon mode with FILL and EDGE
ignored	ignored	12	Draw a FRAME
pen number	ignored	13	Area pen value
red value	green value	14	Color
blue value	ignored	15	Value
ignored	ignored	>15	Ignored

For a detailed description of these parameters, see IPLOT, PLOT, RPLOT, or SYMBOL in the *BASIC Language Reference* manual.

The AREA INTENSITY statement is how you get shades of gray on a black-and-white CRT whose electron gun is either fully on or completely off. You can get seventeen shades of gray. This is done through a process called **dithering**. Dithering is accomplished through selecting small groups of pixels[1], a four-by-four square of them on the Series 200/300 computers. Various pixels in the dithering box are turned on and off to arrive at an "average" shade of gray. There are only seventeen possible shades because out of sixteen pixels (the 4×4 box), you can have none of them on, one of them on, two of them on, and so forth, up to all sixteen of them on. And it makes no difference *which* pixels are on; they are chosen to minimize the striped or polka-dotted pattern inherent to a dithered image.

For more detail on the AREA INTENSITY and other color-related statements, see the "Color Graphics" chapter.

Translating and Rotating a Drawing

Often, there is an application where a segment of a drawing must be replicated in many places; the same sub-picture needs to be drawn many times. Using the PLOT statement, it is possible but rather tedious to do. There is another statement called RPLOT, which draws a figure relative to a point of your choice. RPLOT means *R*elative *PLOT*, and it causes a figure to be drawn relative to a previously-chosen reference point. RPLOT's parameters may be two or three scalars, or a two-column or three-column array; the parameters are identical to those of PLOT.

The picture defined by the data given to an RPLOT statement is drawn relative to a point called the **current relative origin**. This is *not* necessarily the same as the pen position. The current relative origin is the last point resulting from any one of the following statements:

AXES	DRAW	FRAME	GINIT	GRID
IDRAW	IMOVE	IPLOT	LABEL	MOVE
PLOT	POLYGON	POLYLINE	RECTANGLE	

Typically, a MOVE is used to position the current relative origin at the desired location, then the RPLOT is executed to draw the figure. After the RPLOT statement has executed, the pen may be in a different place, but **the current relative origin has not moved**. Thus, executing two identical RPLOT statements, one immediately after the other, results in the figure being drawn precisely on top of itself.

[1] The word "pixel" is a blend of the two words "picture element," and it is the smallest addressable point on a plotting surface. A Model 236 computer has 512×390-pixel resolution; thus there can be no more than 512 dots drawn on any row, or *scan line*, of the CRT, or 390 dots drawn in any column.

A figure drawn with RPLOT can be rotated by using the PIVOT or PDIR statement before the RPLOT. The single parameter for a PIVOT or PDIR is a numeric expression designating the angular distance through which the figure is to be rotated when drawn. This value is interpreted according to the current angular mode: either DEG or RAD.

Here is a program using an RPLOT. It is found on the *Manual Examples Disc* under the file name "Rplot". Various figures are defined with DATA statements: a desk, a chair, a table, and a bookshelf. The program displays a floor layout. Here again, the "end polygon mode" codes (the `0,0,7`s in the desk and chair definitions) are unnecessary; when a polygon mode starts, any previous one ends by necessity.

```
100   CLEAR SCREEN                      ! Clear the alpha display
110   OPTION BASE 1                     ! Make arrays start at one
120   DIM Room(10,3),Desk(18,3),Chair(14,3),Bookshelf(4,3),Table(4,3)
130   READ Room(*),Desk(*),Chair(*),Bookshelf(*),Table(*)
140   GINIT                             ! Initialize various graphics parameters
150   PLOTTER IS CRT,"INTERNAL"         ! Use the internal screen
160   GRAPHICS ON                       ! Display the graphics screen
170   SHOW 0,120,-10,100                ! Need isotropic units for a map
180   PLOT Room(*)                      ! Draw outline of room
190   DEG                               ! Set degrees mode for angles
200   READ Object$                      ! What to draw?
210   WHILE Object$<>"***STOP***"       ! Until done...
220     READ X,Y,Angle                  ! Read where and at what angle
230     MOVE X,Y                        ! Move in unrotated coordinates
240     PIVOT Angle                     ! Set rotation for RPLOTs
250     SELECT Object$
260       CASE "Desk"
270         AREA INTENSITY .125,.125,.125     ! 87.5% gray: dark gray
280         RPLOT Desk(*)
290       CASE "Chair"
300         AREA INTENSITY .5,.5,.5     ! 50% gray: half-and-half
310         RPLOT Chair(*)
320       CASE "Bookshelf"
330         RPLOT Bookshelf(*),EDGE
340       CASE "Table"
350         AREA INTENSITY 0,0,0        ! 100% gray scale: Black
360         RPLOT Table(*),FILL,EDGE
370     END SELECT
380     READ Object$
390   END WHILE
400 Room:     DATA 0,60,-2,   0,100,-1,  120,100,-1, 120,30,-1
```

```
410           DATA 120,20,-2, 120,0,-1,  40,0,-1,    40,25,-1
420           DATA 0,25,-1,   0,50,-1
430 Desk:     DATA 0,0,11,  0,0,-2,    20,0,-1,     20,-10,-1,  0,-10,-1,  0,0,7
440           DATA 0,0,10,  2,-10,-2,  2,-10.5,-1,  3,-10.5,-1, 3,-10,-1,  0,0,7
450           DATA 0,0,10,  17,-10,-2, 17,-10.5,-1, 18,-10.5,-1,18,-10,-1, 0,0,7
460 Chair:    DATA 0,0,11,  -3,9,-2,   3,9,-1,      4,8,-1,     3,2,-1
470           DATA -3,2,-1, -4,8,-1,   0,0,7
480           DATA 0,0,10,  -4,1,-2,   4,1,-1,      4,0,-1,     -4,0,-1,   0,0,7
490 Bookshelf:DATA 0,0,-2,  20,0,-1,   20,-4,-1,    0,-4,-1
500 Table:    DATA 0,0,-2,  25,0,-1,   25,-12,-1,   0,-12,-1
510 Objects:  DATA Chair,    14,75,90    ! \
520           DATA Desk,     1,65,90     !  >  Upper left corner of the room
530           DATA Table,    1,99,0      ! /
540           DATA Bookshelf,27,99,0     !/
550           DATA Chair,    66,44,30    ! \
560           DATA Desk,     50,50,30    !  >  Center of the room
570           DATA Chair,    45,65,210   ! /
580           DATA Desk,     60,58,210   !/
590           DATA Bookshelf,41,5,0      !\
600           DATA Bookshelf,62,5,0      ! >   Bottom center of room
610           DATA Bookshelf,83,5,0      !/
620           DATA Chair,    6,26,0      !\
630           DATA Chair,    16,26,0     ! \
640           DATA Chair,    26,26,0     !  >  Four chairs by west door
650           DATA Chair,    36,26,0     ! /
660           DATA Chair,    63,96,220   ! \
670           DATA Chair,    85,83,3     !  > Four chairs by northeast tables
680           DATA Chair,    112,83,0    ! /
690           DATA Chair,    100,83,355  !/
700           DATA Table,    68,99,0     ! \
710           DATA Table,    94,99,0     !  >  Two tables in upper right
720           DATA Chair,    105,50,270  ! \
730           DATA Desk,     119,60,270  !  >  Desk and chair by east door
740           DATA ***STOP***
750   END
```

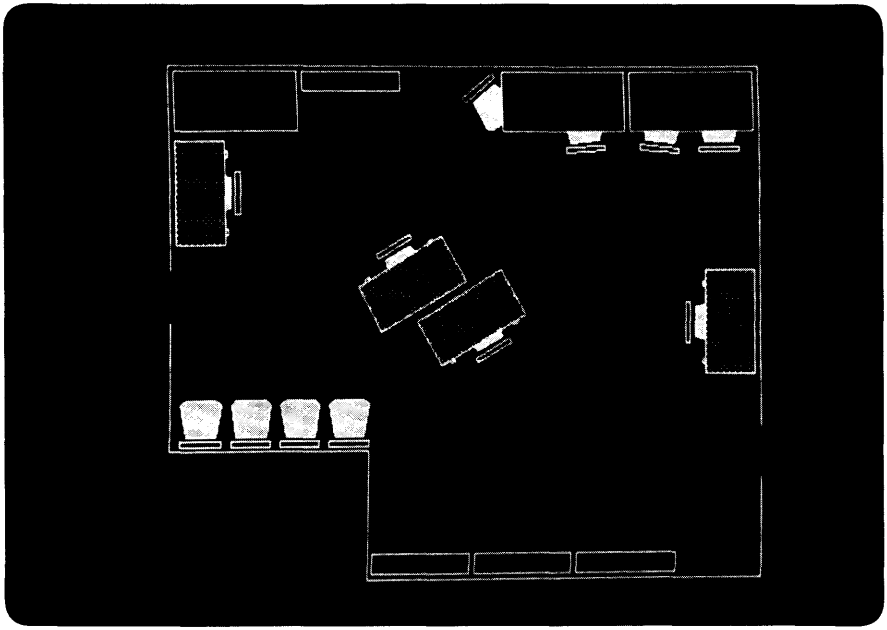


Figure 2-15. Relative Plotting of a Floor Layout (Rplot)

There are two points of interest in this program. First, notice that you can specify the EDGE and/or FILL parameters in the RPLOT statement itself, in addition to in the array. (FILLs and EDGEs are specified in the array by having a 6, a 10, or an 11 in the third column of the array.) If FILL and/or EDGE are specified both in the PLOT statement and in the data, and the instructions differ, the value in the data replaces the FILL or EDGE keyword on the statement.

The second interesting point is that some of the chairs appear to be *under* the desks and tables; that is, parts of several chairs are hidden by other pieces of furniture. This is accomplished by drawing the chair, and then drawing the desk or table partially over the chair, and filling the desktop or tabletop with its own fill pattern, which may be black.

Incremental Plotting

Incremental plotting is similar to relative plotting, except that the origin—the point considered to be 0,0—is moved every point. Every time you move or draw to a point, the origin is immediately moved to the new point, so the next move or draw will be with respect to that new origin.

There are three incremental plotting statements available: IPLOT, which has the same parameters as PLOT and RPLOT; and IMOVE and IDRAW, which have the same parameters as MOVE and DRAW, respectively.

Below is an example program using IPLOTs. It reads data from data statements describing the outlines of certain letters of the alphabet, and then plots them. (See "Iplot" on the *Manual Examples Disc*.)

```
100     CLEAR SCREEN                ! Clear the alpha display
110     OPTION BASE 1               ! Make the arrays start at 1
120     DIM Array(20,3)             ! Set aside space for the array
130     GINIT                       ! Initialize various graphics parameters
140     PLOTTER IS CRT,"INTERNAL"   ! Use the internal screen
150     GRAPHICS ON                 ! Turn on graphics screen
160     SHOW 1,35,-15,15            ! Isotropic scaling
170     FOR Letter=1 TO 4           ! Four letters total
180       READ Points               ! How many points in this letter?
190       REDIM Array(Points,3)     ! Adjust the array size accordingly
200       READ Array(*)             ! Read the correct number of points
210       MOVE Letter*6,0           ! Move to lower-left corner of letter
220       IPLOT Array(*)            ! Draw letter
230     NEXT Letter                 ! et cetera
240 F:  DATA 10,   0,5,-1,      5,0,-1,      0,-1,-1,      -4,0,-1,      0,-1,-1
250     DATA       3,0,-1,      0,-1,-1,     -3,0,-1,      0,-2,-1,      -1,0,-1
260 L:  DATA 6,    0,5,-1,      1,0,-1,      0,-4,-1,      4,0,-1,       0,-1,-1
270     DATA       -5,0,-1
280 A:  DATA 12,   2,5,-1,      1,0,-1,      2,-5,-1,      -1,0,-1,      -.4,1,-1
290     DATA       -2.2,0,-1,   -.4,-1,-1,   -1,0,-1,      1.8,2,-2,     .7,2,-1
300     DATA       .7,-2,-1,    -1.4,0,-1
310 X:  DATA 12,   1.9,2.5,-1,  -1.9,2.5,-1, 1,0,-1,       1.5,-2,-1,    1.5,2,-1
320     DATA       1,0,-1,      -1.9,-2.5,-1,1.9,-2.5,-1,  -1,0,-1,      -1.5,2,-1
330     DATA       -1.5,-2,-1,  -1,0,-1
340     END
```

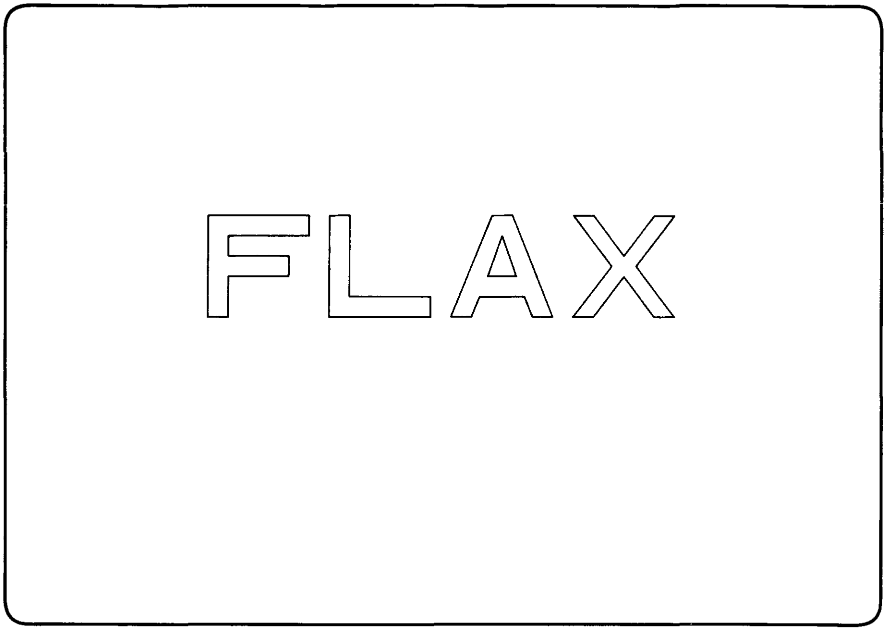


Figure 2-16. Incrementally Plotting Letters (Iplot)

Drawing Polygons

When you want a regular[1] polygon, or a part of one, drawn on the screen, there are two statements which will help. The first is called POLYGON.

One attribute of POLYGON is that *it forces polygon closure*, that is, the first vertex is connected to the last vertex, so there is always an inside and an outside area[2]. There are two final keywords which may be included in a POLYGON statement, and they are FILL and EDGE. FILL causes the interior of the polygon or polygon segment to be filled with the current fill color as defined by AREA PEN, AREA COLOR, or AREA INTENSITY. FILL specified without EDGE causes the interior of the polygon to be indistinguishable from the edge. EDGE causes the edges of the polygon go be drawn using the current pen and line type. If both FILL and EDGE are specified (and FILL must be first), the interior will be filled, then the edge will be drawn. If neither FILL nor EDGE is specified, EDGE is assumed. On an HPGL plotter, only EDGE works.

---

[1] In this discussion, polygons drawn when anisotropic units are in effect will also be considered "regular". Anisotropic units will cause stretching or compression of the polygons in the X or Y direction.

[2] Technically, this is true even for the degenerate case of drawing only one side of a polygon, in which case a "single" line results. This is actually two lines, from the first point to the last point, and back to the first.

Polygons can be rotated by specifying a non-zero value in a PIVOT or PDIR statement before the POLYGON statement is executed. Also, a PDIR statement can be used to specify the angle of rotation. PDIR works with IPLOT, RPLOT, POLYLINE, POLYGON, and RECTANGLE. The rotation occurs about the origin of the figure. For example, PDIR 15 would rotate a figure 15 units (degrees, radians).

The shape of the polygon is affected by the viewing transformation specified by SHOW or WINDOW. Therefore, anisotropic scaling causes the polygon to be stretched or compressed along the X and Y axes.

The pen status also affects the way a POLYGON statement works. If the pen is up at the time POLYGON is specified, the first vertex specified is connected to the last vertex specified, *not* including the center of the polygon, which is the current pen position. If the pen is down, however, the center of the polygon is also included in it. Thus, for piece-of-pie shaped polygon segments, like are used in pie charts, cause the pen to be down before the POLYGON statement is executed.

After POLYGON has executed, the current pen position is in the same position it was before the statement was executed, and the pen is up.

But I don't want polygon closure...

There is another statement called POLYLINE which acts much in the same way as POLYGON, except it does not connect the last vertex to the first vertex; it does not *close* the polygon. Obviously, then, since the polygon is not closed, there is no "inside" or "outside," hence it is meaningless to say FILL or EDGE.

As in the case of POLYGON, a PIVOT or PDIR statement prior to execution of POLYLINE will cause the figure to be rotated. Anisotropic scaling will cause stretching or compression along the axes, and if the pen is down prior to invocation of the statement, a line will be drawn from the center to the first perimeter point.

After POLYLINE has executed, the current pen position is in the same position it was before the statement was executed, and the pen is up.

Following is a program which demonstates the use of POLYGON, POLYLINE, PLOT, RPLOT, polygon filling, and gray-shading. The program may be loaded from file "Scenery" on the *Manual Examples Disc.*

```
100   CLEAR SCREEN                                   ! Clear the alpha display
110   OPTION BASE 1                                  ! Arrays start at 1.
120   DIM Horizon(20,2),Tree(24,2),Tree2(24,2)       ! For PLOT, RPLOT
130   GINIT                                          ! Initialize graphics parameters
140   PLOTTER IS CRT,"INTERNAL"                      ! Use the internal screen
150   GRAPHICS ON                                    ! Turn on graphics screen
160   WINDOW 0,511,0,389                             ! 1 UDU = 1 pixel
170   RANDOMIZE 123456789                            ! "Looks better" than default
180    ! Draw sunrise-----------------------------------------------------------
190   Sun_diameter=30                                ! Diameter of outer layer
200   Sun_delta=6                                    ! Shrinkage of each brightness
210   MOVE 256,190                                   ! Center of sun
220   FOR I=1/16 TO 1 STEP 1/16                      ! All non-black gray shades
230     AREA INTENSITY I,I,I                         ! Define dithered gray shade
240     POLYGON Sun_diameter+(16-16*I)*Sun_delta,30,FILL   ! Draw sun
250   NEXT I                                         ! and so forth
260    ! Draw horizon-----------------------------------------------------------
270   Horizon(1,1)=0                                 ! \ Lower left corner of screen,
280   Horizon(1,2)=0                                 ! / for blacking bottom of sun
290   Dx=511/(20-3)                                  ! Delta X for horizon
300   X=-Dx                                          ! Starting point for X
310   FOR I=2 TO 19                                  ! All except end points
320     X=X+Dx                                       ! Increment X
330     Horizon(I,1)=X                               ! Put it in the array
340     Horizon(I,2)=185+RND*10                      ! Random height for roughness
350   NEXT I                                         ! and so forth
360   Horizon(20,1)=511                              ! \ Lower right corner of screen
370   Horizon(20,2)=0                                ! / for blacking bottom of sun
380   AREA INTENSITY 0,0,0                           ! Black
390   PLOT Horizon(*),FILL                           ! Erase bottom of sun
400   PENUP                                          ! PLOT left pen down
410   FOR I=2 TO 20-1                                ! \   Draw the horizon polygon,
420     PLOT Horizon(I,1),Horizon(I,2)               !  >  but don't include first
430   NEXT I                                         ! /   and last points (corners).
440    ! Draw clouds------------------------------------------------------------
450   WINDOW -2,2,-15,15                             ! Anisotropic scaling
460   AREA INTENSITY .25,.25,.25                     ! 25% gray shade
470   FOR I=1 TO 10                                  ! 10 ellipses
480     MOVE RND^.8*4-2,RND*8                        ! Random position
490     POLYGON RND*.8,FILL                          ! random size, fill it
500   NEXT I                                         ! and so forth
510   WINDOW 0,511,0,389                             ! Back to 1 UDU = 1 pixel
520    ! Draw birds-------------------------------------------------------------
530   DEG                                            ! Angular mode: Degrees
540   Phi=70                                         ! Arc subtended by each wing
550   FOR Bird=1 TO 10                               ! Ten birds enough
560     Position_angle=RND*360                       ! Bird's direction from 100,300
570     Distance=SQR(RND)*70                         ! Bird's distance from 100,300
```

```
580     X=100+Distance*COS(Position_angle)       ! Bird's actual X position
590     Y=300+Distance*SIN(Position_angle)       ! Bird's actual Y position
600     Theta=RND*20-10                          ! Bird's tilt
610     R=RND*10+10                              ! Radius of arcs of birds' wings
620     Left_angle=180+(90-Phi/2)+Theta          ! Direction of left arc's center
630     X2=X+R*COS(Left_angle)                   ! Center of left wing's arc (X)
640     Y2=Y+R*SIN(Left_angle)                   ! Center of left wing's arc (Y)
650     PIVOT 0                                  ! Unrotated coords for MOVE
660     MOVE X2,Y2                               ! Left arc's center
670     PIVOT Theta+90-Phi/2                     ! Rotated coords for POLYLINE
680     POLYLINE R,60,60*Phi/360                 ! Left wing's arc
690     Right_angle=Theta-90+Phi/2               ! Right arc's center's direction
700     X3=X+R*COS(Right_angle)                  ! Center of right wing's arc (X)
710     Y3=Y+R*SIN(Right_angle)                  ! Center of right wing's arc (Y)
720     PIVOT 0                                  ! Unrotated coords for MOVE
730     MOVE X3,Y3                               ! Right arc's center
740     PIVOT Theta+90-Phi/2                     ! Rotated coords for POLYLINE
750     POLYLINE R,60,60*Phi/360                 ! Right wing's arc
760   NEXT Bird                                  ! and so forth
770   PIVOT 0                                    ! Back to normal for trees
780    ! Draw trees-------------------------------------------------------------
790   AREA INTENSITY .5,.5,.5                    ! 50% gray
800   Tree(1,1)=-.5                              ! \
810   Tree(1,2)=0                                !  \  Define by hand the trunk
820   Tree(2,1)=-.5                              !  /  of the tree
830   Tree(2,2)=1                                ! /
840   FOR I=3 TO 12 STEP 2                       ! \
850     Tree(I,1)=-((13-I)/4)                    !  \
860     Tree(I,2)=(I-1)/2                        !   \  Define programmatically
870     Tree(I+1,1)=(Tree(I,1)+.5)/2             !   /  the branches of the trees
880     Tree(I+1,2)=Tree(I,2)+1                  !  /
890   NEXT I                                     ! /
900   FOR I=13 TO 24                             ! \   The right half of the tree
910     Tree(I,1)=-Tree(25-I,1)                  !  \  (and thus the tree array,)
920     Tree(I,2)=Tree(25-I,2)                   !  /  is the mirror image of the
930   NEXT I                                     ! /   left half.
940   Y=180                                      ! Starting value
950   WHILE Y>10                                 ! For a few iterations...
960     FOR I=1 TO Y^(Y/180)/2                   ! No. of trees dependent upon Y
970       Y2=RND*20                              ! Random variation
980       MOVE RND*511,Y+Y2-15                   ! Bottom of center of tree
990       Size=(200-(Y+Y2))*.1                   ! Size of tree dependent upon Y
1000      MAT Tree2= Tree*(Size)                 ! Scale tree appropriately
1010      RPLOT Tree2(*),FILL                    ! FILL, but don't EDGE
1020    NEXT I                                   ! and so forth
1030    Y=Y*.8                                   ! Go lower on the screen
1040  END WHILE                                  ! for a while...
1050  END
```

Figure 2-17. Using POLYGON and POLYLINE (Scenery)

Points of note in this program:

1. The sunrise was created with graduated gray shades in successively smaller "circles" (actually 30-sided polygons).
2. The horizon was created by defining a rough edge on the top half of a polygon which blacked out the bottom section of the screen. This covered up the bottom of the sun. The white line of the horizon was simple plotting of the horizon array *without the first and last points.* We didn't want the lower corners of the screen to be included.

3. The clouds were created by plotting "circles" after having invoked anisotropic units; thus long, thin ellipses resulted.
4. The seagulls were created by drawing two arcs with POLYLINE. An arc is created by defining an N-sided polygon and drawing less-than-N sides. Note that PIVOT was used to cause the starting angle of the arcs to be other than straight to the right.
5. The trees were created by defining an array whose left side is a mirror image of the right side. The array is centered around zero in the X direction to allow for scaling of the tree simply by multiplying the array by a constant. RPLOT was used to place the trees in their various positions.

Rectangles

One of the most-used polygons in computer graphics is the rectangle. You can cause a rectangle to be drawn by moving to the point at which you want one of the corners to be and then specifying which directions to proceed from there, first in the X direction, then in the Y direction. Which corner of the rectangle ends up at the current pen position depends on the signs of the X and Y parameters. For example, if you want a rectangle whose lower left corner is at 3,2 and which is 4 units wide and 5 units high, there are four ways you could go about it:

```
MOVE 3,2              (Reference point is the lower left corner)
RECTANGLE 4,5
```

or

```
MOVE 7,2              (Reference point is the lower right corner)
RECTANGLE -4,5
```

or

```
MOVE 3,7              (Reference point is the upper left corner)
RECTANGLE 4,-5
```

or

```
MOVE 7,7              (Reference point is the upper right corner)
RECTANGLE -4,-5
```

Again, you can specify FILL, EDGE, or both. FILL will cause the rectangle to be filled with the current fill color as specified by AREA PEN, AREA COLOR, or AREA INTENSITY. EDGE causes the edge of the rectangle to be drawn in the current pen color and line type. If both are specified, FILL must be specified first, and if neither is specified, EDGE is assumed. The current pen position is not changed by this statement, and pen status prior to execution makes no difference in the resulting rectangle.

User-Defined Characters

For many special-purpose programs, there is a drastic shortage of characters that can be displayed on the screen[1]. Greek letters—π, Δ, Σ, and so forth—are quite often needed for mathematics-intensive communication as well as many non-alphabetic symbols like $\surd$, ∞, and $\pm$. To alleviate this shortage of symbols, the SYMBOL statement allows you to draw any definable character. In function, it is similar to PLOT using an array, except the figure drawn by SYMBOL is subject to the three transformations which deal with character labelling: CSIZE, LDIR, and LORG.

The first argument needed by the SYMBOL statement is the array containing the instructions on what to draw. As in PLOT, this array may either have two or three columns. If the third column does not exist, it is assumed to be +1 for every row of the array. If it does exist, the valid values for the third-column entries are identical to those for PLOT, RPLOT, and IPLOT when using an array. The possible values for the third column are listed again here for your convenience.

[1] User-defined alpha character fonts are available on bit-mapped alpha displays. See the "Communicating with the Operator" chapter of *BASIC Programming Techniques*.

Table 2-3. Pen Control when Plotting Entire Arrays (Repeated)

Column 1	Column 2	Operation Selector	Meaning
X	Y	−2	Pen up before moving
X	Y	−1	Pen down before moving
X	Y	0	Pen up after moving (Same as +2)
X	Y	1	Pen down after moving
X	Y	2	Pen up after moving
pen number	ignored	3	Select pen
line type	repeat value	4	Select line type
color	ignored	5	Color value
ignored	ignored	6	Start polygon mode with FILL
ignored	ignored	7	End polygon mode
ignored	ignored	8	End of data for array
ignored	ignored	9	NOP (no operation)
ignored	ignored	10	Start polygon mode with EDGE
ignored	ignored	11	Start polygon mode with FILL and EDGE
ignored	ignored	12	Draw a FRAME
pen number	ignored	13	Area pen value
red value	green value	14	Color
blue value	ignored	15	Value
ignored	ignored	>15	Ignored

For more detail on the meaning of these values, see the *BASIC Language Reference* manual.

Moves and draws specified in an array to be used in a SYMBOL statement are defined in the **symbol coordinate system**. This coordinate system is a character cell, as defined earlier in the chapter—a 9×15 rectangle. Figures drawn in this coordinate system may be filled or edged or both. The FILL and EDGE keywords may appear in the SYMBOL statement itself, or they may be specified in the data array. If FILL and/or EDGE are specified in both places, the instruction in the data array overrides that of the statement.

One interesting feature of this statement is that values *outside* the character cell boundaries are valid. Thus, you can define characters that are several lines high, several characters wide, or both. This feature is used in the following program. The SYMBOL statement, by virtue of its syntax, can only be used for one User-Defined Character (UDC) at a time; the pen must be moved to the new position before each character. Therefore, UDCs cannot be embedded in a string of text. If the situation remained this way, the utility of the SYMBOL statement would be limited by its cumbersome implementation. The following program makes UDCs much easier to use. It is a special-purpose program which calls two general-purpose subprograms. The first subprogram (`New_udc`) is called to define a new UDC. Its parameters are: 1) the character to be replaced by the UDC, and 2) the array defining the character. The second subprogram (`Label`) is called after all desired UDCs have been defined. This allows text to be labelled (written in graphics mode) intermixing ASCII characters with user-defined characters at will. As mentioned above, all user-defined characters are affected by CSIZE, LDIR and LORG, so no matter how the label is being written, the UDCs will act properly.

Four characters are defined below: a Greek capital sigma (the summation sign), infinity (a figure eight who's expired), a fat arrow pointing to the right, and a large box. Note that the box is three characters wide; it is perfectly legal to have points going outside the 9×15 bounds of the character cell. This program may be loaded from the file "Symbol" on the *Manual Examples Disc.*

```
100     OPTION BASE 1
110     COM /Udc/ Old_chars$[20],Size(20),Chars(20,30,3)
120     REAL Sigma(7,3),Infinity(16,3),Arrow(9,3),Box(12,3)
130     READ Sigma(*),Infinity(*),Arrow(*),Box(*)
140 Sigma:   DATA 7,5,-2,      7,4,-1,      1,4,-1,      5.5,8.5,-1
150          DATA 1,13,-1,     7,13,-1,     7,12,-1
160 Infinity:DATA 4,9,-2,      5,10,-1,     6,10,-1,     7,9,-1
170          DATA 7,8,-1,      6,7,-1,      5,7,-1,      4,8,-1
180          DATA 4,9,-1,      3,10,-1,     2,10,-1,     1,9,-1
190          DATA 1,8,-1,      2,7,-1,      3,7,-1,      4,8,-1
200 Arrow:   DATA 0,0,6,       4,4,-2,      7,8,-1,      4,12,-1
210          DATA 4,10,-1,     1,10,-1,     1,6,-1,      4,6,-1
220          DATA 0,0,7
230 Box:     DATA 0,0,6,       3,0,-2,      27,0,-1,     27,15,-1
240          DATA 0,15,-1,     0,0,-1,      3,0,-1,      3,3,-1
250          DATA 24,3,-1,     24,12,-1,    3,12,-1,     0,0,7
260     Old_chars$=""      ! In case anything is left in COM from the last run...
270     New_udc(CHR$(168),Sigma(*))     !  \
280     New_udc(CHR$(169),Infinity(*))  !   >  Replace unneeded characters with
290     New_udc(CHR$(170),Arrow(*))     !  /   User-Defined Characters
300     New_udc(CHR$(171),Box(*))       ! /
```

```
310    KEY LABELS OFF
320    CLEAR SCREEN                              ! Clear the alpha display
330    GINIT
340    PLOTTER IS CRT,"INTERNAL"         ! Use the internal screen
350    GRAPHICS ON
360    SHOW 0,10,-.5,10
370    DEG
380    FOR Csize=10 TO 2 STEP -1
390      CSIZE Csize
400      FOR Ldir=0 TO 90 STEP 90
410        LORG 2
420        LDIR Ldir
430        MOVE 10-Csize,10-Csize
440        Label(" Chars: '"&CHR$(168)&CHR$(169)&CHR$(170)&CHR$(171)&"  '")
450      NEXT Ldir
460    NEXT Csize
470    END
480    ! ***********************************************************************
490 New_udc:    SUB New_udc(Char$,Array(*))
500    !  This allows up to twenty new characters to be defined, each having up
510    !  to thirty elements (rows in the array) for definition.
520    OPTION BASE 1
530    COM /Udc/ Old_chars$[20],Size(20),Chars(20,30,3)
540    IF LEN(Old_chars$)=20 THEN
550      PRINT "User-defined Character table full."
560    ELSE  ! (still room)
570      Pos=LEN(Old_chars$)+1
580      Old_chars$[Pos]=Char$
590      Size(Pos)=SIZE(Array,1)
600      FOR Row=1 TO Size(Pos)
610        FOR Column=1 TO 3
620          Chars(Pos,Row,Column)=Array(Row,Column)
630        NEXT Column
640      NEXT Row
650    END IF  ! (room left?)
660    SUBEND
670    ! ***********************************************************************
680 Label:     SUB Label(Text$)
```

```
690   ! This prints a character string at the current pen position and using
700   ! the current LORG, LDIR and CSIZE. The LORG will need to be redeclared
710   ! upon returning to the calling context, as this routine needs LORG 1 if
720   ! the text is longer than one character.
730   OPTION BASE 1
740   COM /Udc/ Old_chars$[20],Size(20),Chars(20,30,3)
750   REAL Array(31,3)
760   FOR Char=1 TO LEN(Text$)
770     IF Char=2 THEN LORG 1    ! Necessary when doing one character at a time
780     Char$=Text$[Char;1]
790     Pos=POS(Old_chars$,Char$)            ! Is this to be replaced by a UDC?
800     IF Pos THEN
810       REDIM Array(Size(Pos),3)                    ! \
820       FOR Row=1 TO Size(Pos)                      !  \   Take a slice out
830         FOR Column=1 TO 3                         !   >  of the 3D array
840           Array(Row,Column)=Chars(Pos,Row,Column) !  /   and put it in the
850         NEXT Column                               ! /    2D array for
860       NEXT Row                                    !/     SYMBOL.
870       WHERE X,Y
880       SYMBOL Array(*)
890       MOVE X,Y
900       LABEL USING "#,K";" "  ! Tell the computer to update the pen position
910     ELSE  ! (regular character)
920       LABEL USING "#,K";Char$
930     END IF  ! (this character been redefined?)
940   NEXT Char
950   SUBEND
```

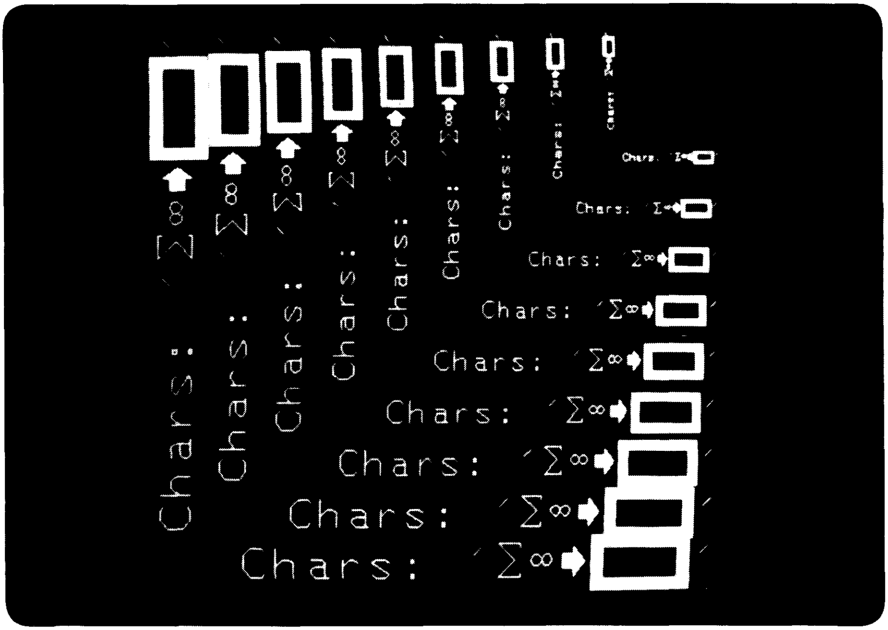


Figure 2-18. Implementing User-Defined Characters (Symbol)

Of course, the limits (twenty UDCs and thirty rows maximum) may be reduced or expanded to fit whatever purpose for which you need it. Note that in lines 180 through 210 of the program, characters 168 through 171 were replaced by the four UDCs. There is nothing magical about these four characters. The characters replaced could have been any characters between 0 and 255, and they need not be consecutive.

Also note that in line 450 of the program, there are two spaces after the CHR$(171). This is because character 171 was replaced by the box, which was three character cells wide. The two extra spaces prevent the right two-thirds of the box from being overwritten by whatever is to be labelled after it.

Multi-Plane Bit-Mapped Displays

When using multi-plane (color) bit-mapped displays, BASIC provides the ability to specify which planes to write-enable for alpha and graphics and which planes to display. This feature provides for several useful features on bit-mapped displays. But before we look at the uses, let's look at what these masks are.

There are four main areas of interest that we'll cover:

- The graphics write-enable mask,
- The display-enable mask,
- The alpha write-enable mask (this is similar in structure and operation to the graphics write-enable mask, so little will be said here about it), and
- Interactions between the alpha masks and graphics masks.

The Graphics Write-Enable Mask

First, we'll look at the **graphics write-enable mask**. As its name (partially) implies, its function is to specify which frame buffer planes are to be written to by graphics operations.

For the purposes of illustration, assume that your machine has four planes in the frame buffer. Suppose you want to set the graphics write-enable mask such that graphics operations use only the first two planes of the frame buffer. This is effected by setting the first element of an integer array to the value desired, and then executing the `GESCAPE` with operation selector 7:

```
INTEGER Graphics_masks(0:0)
Graphics_masks(0)=IVAL("0011",2)      ! Set graphics write mask: planes 1&2
GESCAPE CRT,7,Graphics_masks(*)       ! Set write mask; display mask is unchanged
```

If, as in this example, the graphics write-enable mask has the value `0011` (in decimal, the value is 3, but the masks will be shown in binary for clarity), this indicates that only planes 1 and 2 of the frame buffer will be used for graphics write operations. For example, drawing a line can change bits only in planes 1 and 2.

A graphics write-enable mask of `0011` also implies that there are only four color-map "pens" available for use by graphics operations: 0 through 3.

The Graphics Display-Enable Mask

In addition to the graphics write-enable mask, there is a graphics display-enable mask. The difference between them is indicated by their respective names: the former specifies which planes are actually modified by graphics operations (regardless of whether or not they are displayed), and the latter specifies which planes can be seen by the user (regardless of whether or not anything has been or can be written to them).

Suppose you want to set the graphics display-enable mask such that only the contents of the first two planes of the frame buffer are visible to the user. This is effected by setting the *second* element of an integer array to the value desired, and then executing GESCAPE with operation selector 7:

```
INTEGER Graphics_masks(0:1)
Graphics_masks(0)=<some value>
Graphics_masks(1)=IVAL("0011",2)     ! Set display-enable mask: planes 1&2
GESCAPE CRT,7,Graphics_masks(*)      ! Set masks
```

Although for many operations, the graphics write-enable mask and the graphics display-enable mask will have the same value, they need not be the same. In fact, there are many instances where they will be different. In many of these cases, one or both of the graphics masks will change regularly as the program executes.

The Alpha Masks

The alpha write-enable mask and display-enable mask do the same jobs for the alpha display as their graphics counterparts do for the graphics display.

Note

When the PLOTTER IS device *is the same as* the ALPHA crt device, the alpha display mask *is* the graphics display mask. Even though they are the same entity, it can be accessed either by GESCAPE CRT,7 or CONTROL CRT,20 (see below).

To set and read the alpha write-enable mask, use the following statements:

```
SET ALPHA MASK IVAL("1100",2)     ! Set alpha WRITE-enable mask: planes 3&4
STATUS CRT,18;Alpha_writemask     ! Read alpha WRITE-enable mask
```

To set and read the alpha display-enable mask, use the following statements:

```
SET DISPLAY MASK IVAL("1100",2)  ! Set alpha DISPLAY-enable mask: planes 3&4
STATUS CRT,20;Alpha_disp_mask    ! Read alpha DISPLAY-enable mask
```

Interactions Between Alpha and Graphics Masks

The alpha and graphics write-enable masks both default at power-up and SCRATCH A to all planes in the machine. Thus (again, assuming you have a four-plane machine), both write-enable masks would be 1111. All four planes are at once *both* alpha planes and graphics planes. One implication to this is that when you write graphical information, the alpha display may be modified/overwritten, and vice versa.

Alternately, if you set the alpha write-enable mask to 0011 (alpha uses planes 1 and 2), and the graphics display-enable mask to 1000 (graphics uses plane 4), plane 3 is neither alpha nor graphics.

This is a departure from older machine architectures where there was a fixed alpha area and a fixed graphics area, and in which "alpha" equals "not graphics" and "graphics" equals "not alpha." Now, planes can be "both alpha and graphics" or "neither alpha nor graphics," and you can choose where to put them.

When a plane is designated as both an alpha plane and a graphics plane (default state), alpha and graphics share the bit-mapped plane so that writing to the display overwrites existing information. A plane designated as a graphics plane *only* can be written to by graphics statements only, such as DRAW, LABEL, but not with alpha statements. Similarly, an alpha-only plane can be written to with statements such as PRINT and DISP, but not with graphics statements[1].

One use of this feature is to simulate separate alpha and graphics planes for compatibility with older systems which did not have bit-mapped alpha. Assuming you have a four-plane display, you could designate planes 1 through 3 for graphics, and plane 4 for alpha. This gives you eight pure graphics colors, and a single alpha color. This gives you some of the capabilities of a separate alpha/graphics system on bit-mapped hardware:

- Turning alpha and graphics on and off independently,
- Dumping the graphics display or the alpha display independently[2].
- Scrolling alpha without scrolling graphics along with it.

---

[1] There is also an interaction with the color map setup when executing PLOTTER IS: PENs with bits in non-graphics planes are not updated when the color map is initialized. The graphics write-enable mask also affects GSTORE and GLOAD; see the *BASIC Language Reference* for details.

[2] It has always been the case that you could dump alpha when using bit-mapped displays. However, this capability is afforded by the presence of an "alpha buffer," a spare storage place for all alpha information. This enables alpha to be dumped to a printer which does not have raster graphics capabilities.

There is one tricky condition that may occur if you're not careful. Suppose your program has executed ALPHA PEN 1, and at some point it changes the alpha write-enable mask to 1100. All of a sudden, the output stops appearing, the run light remains on, and the "live" keyboard appears dead. By all appearances, your machine is hung. This is not the case, however.

What actually happened was this. Before the write-enable mask was changed, everything was going as expected. When you changed the write-enable mask to 1100, you caused the machine to only turn on pixels when using pens whose numbers had some bits in common with the write-enable mask, now 1100. The only pens which satisfy this condition are pens 4 through 15. In other words, nothing appeared when you were "writing" because on one hand you said "write pixels in planes 3 and 4 only," and on the other hand you said "I am going to write with a pen which doesn't turn on pixels in planes 3 and 4."

The following program illustrates the concept of simulating separate alpha and graphics. (You can also use the SEPARATE ALPHA FROM GRAPHICS and MERGE ALPHA WITH GRAPHICS statements to accomplish these tasks; however, this program shows the intermediate steps in how the alpha *write*-enable and *display*-enable masks interact). The program will write some text which will be visible. Next, it will change the display mask so the alpha display mask and the current pen are disjoint. After writing a bit more, the program will pause, so you can see just how hung it really looks. When you press [CONTINUE] ([f2] in the System menu of an ITF keyboard), the alpha display mask and the pen will once again come into agreement, and all will be as it should be.

```
10    ! This program demonstrates proper and improper use of alpha masks.
20    ! (Return everything to its default state if the program didn't finish)
30    SET ALPHA MASK IVAL("1111",2)       ! Set alpha write-enable mask.
40    SET DISPLAY MASK IVAL("1111",2)     ! Set alpha display-enable mask.
50    CLEAR SCREEN
60    GINIT
70    PLOTTER IS CRT,"INTERNAL";COLOR MAP
80    ! (Start the demo)
90    ALPHA PEN IVAL("0011",2)            ! Set alpha pen
100   SET ALPHA MASK IVAL("0011",2)       ! Set alpha write-enable mask.
110   SET DISPLAY MASK IVAL("0011",2)     ! Set alpha display-enable mask.
120   PRINT "I'm printing visibly."
130   WAIT 4
140   PRINT "Changing the alpha display mask -- this text will vanish."
150   WAIT 4
160   SET DISPLAY MASK IVAL("1100",2)     ! Set alpha display-enable mask.
170   WAIT 4
180   FOR I=1 TO 10
190     PRINT " I'm printing stuff now but you can't see it."
200   NEXT I
210   SET DISPLAY MASK IVAL("0011",2)     ! Set alpha display-enable mask.
```

```
220   WAIT 1
230   PRINT "The above text just became visible."
240   WAIT 4
250   PRINT "The above text will disappear again and the machine will"
260   PRINT "appear hung until you press CONTINUE."
270   PRINT "(Note the run light stays on after the computer pauses.)"
280   WAIT 9
290   SET DISPLAY MASK IVAL("1100",2)     ! Set alpha display-enable mask.
300   PAUSE
310   SET DISPLAY MASK IVAL("0011",2)     ! Set alpha display-enable mask.
320   WAIT 1
330   PRINT
340   PRINT "The above text just became visible again."
350   WAIT 4
360   SET ALPHA MASK IVAL("1111",2)       ! Set alpha write-enable mask.
370   SET DISPLAY MASK IVAL("1111",2)     ! Set alpha display-enable mask.
380   SET ALPHA PEN IVAL("0001",2)        ! Set alpha pen
390   PRINT "I just set the alpha masks and pen back to normal."
400   END
```

Another use of the graphics write-enable mask is the fast display of multiple pictures. For example, you could:

1. Disable all planes displayed. This makes the screen blank.
2. Enable plane 1 for writing.
3. Create a single-color picture in plane 1.
4. Enable plane 1 for display. Now the picture in plane 1 appears on the screen.
5. Disable plane 1 for writing and enable plane 2.
6. Create a different single-color picture in plane 2.
7. Disable plane 1 for display and enable plane 2. Now the picture in plane 2 appears on the screen.

Cycling through this sort of a loop—flashing consecutive pictures on the screen—simulates a rudimentary animation; i.e., "motion" pictures. Be aware, however, that the drawing speed is much too slow to describe smooth, non-jerky movements.

The selection of graphics planes to be write-enabled and display-enabled is accessed via the GESCAPE statement (in the GRAPHX binary).

Disabling and Enabling Alpha Scrolling

Series 300 computers (and Series 200 Model 237 computer) have alpha and graphics on the same raster (this is called "bit-mapped alpha"). One of the most important implications of this architecture for graphics is that when you scroll alpha information, you also scroll graphics. If you want to disable scrolling, of both alpha and graphics, due to the cursor-control keys (such as [▲] and [▼]), you can execute this statement:

```
CONTROL KBD,16;1
```

If scrolling is currently disabled and you want to re-enable it, execute this statement:

```
CONTROL KBD,16;0
```

On non-bit-mapped alpha displays, `CONTROL KBD 16, 1` will also disable alpha scrolling. Because the alpha and graphics are physically separate, however, graphics will never be affected by alpha scrolling on non-bit-mapped displays.

Using Printers and Plotters 3

Using Printers and Plotters 3

Preceding chapters described how to generate graphics images on your computer's CRT display. In this chapter, we will be discussing the selection of external[1] plotting devices which you can use to generate "hard copy" graphics on paper.

- First we'll show how to **dump graphics** images from a CRT display to a **printer**.
- Then we'll show how to use the PLOTTER IS statement to select HP-IB **plotters**, which may be connected through the built-in HP-IB (Hewlett-Packard Interface Bus) port in the back of your computer. This section also describes how to use "Hewlett-Packard Graphics Language" (HPGL) commands to talk directly to plotters.
- The next section shows how to send plotting commands to a **file**.
- The last part of the chapter describes using **SRM plotter spoolers**.

[1] "External" CRT displays, which may be connected to your computer through 98627A or 98700 display interfaces, are discussed in the "Color Graphics" chapter.

Dumping Raster Images to Printers

After generating an image on your CRT display, you can print it on paper using a printer[1]. This operation is called a *graphics dump* or a *screen dump*. It is accomplished by copying data from the frame buffer to be reproduced on the printer in dot-for-dot fashion.

First, the image must be generated. Any CRT display may be used. The BASIC system "takes a snapshot" of the graphics screen at some point in time, and sends it to a printer. BASIC doesn't care *how* the dots came to be turned on or off. Thus, filled areas can be dumped to the printer; indeed, all CRT graphics capabilities (except color[2]) are available.

Dumping to HP Raster Interface Standard Devices

If your printer conforms to the HP Raster Interface Standard[3], dumping graphics images is trivial. For example:

```
100   DUMP DEVICE IS 701
110   DUMP GRAPHICS
```

or

```
100   DUMP GRAPHICS #701
```

Both of these program segments would take the image in the last specified graphics frame buffer (the internal CRT display by default) and send it to the printer at address 701. (If no source device is specified, the image is taken from the last active CRT or it is the default graphics display.) 701 is the default factory setting for printers. You would probably use the two-statement version in an application where you wish to specify the destination device once, and have it apply to many different DUMP GRAPHICS statements. The one-statement version would probably be used where there are few and isolated DUMP GRAPHICS statements.

1 The subsequent section called "Using SRM Plotter Spoolers" describes how to dump the raster to an SRM spooler.

2 Note that you can do a color dump display to a PaintJet(TM) using the GDUMP_C utility. For more information on this, read the "BASIC Utilities Library" section of the *Installing and Maintaining the BASIC System* manual.

3 In order to determine whether or not your printer conforms to this standard, see your printer's manual or the *Configuraton Reference*.

Dumping from a Color Display

When dumping an image from a color display to a printer, the state of each bit sent to the dump device is calculated by performing an inclusive OR operation on all color-plane bits for each pixel (in other words, a dot is printed if the pen used to write the dot was not equal to 0). Thus, no color information is dumped.

Dumping from a High-Resolution Display

If the source device is a high-resolution display, the image will not fit on one page of most HP printers. In such cases, you should only use the portion of the screen that is dumped to the printer. You can see which portion is not used by drawing a grid, for instance, and then dumping it to the printer. When you see which part is not displayed on the printer, you can change the region of the graphics display (with VIEWPORT, for example) so that you do not use that region.

Using the DUMP GRAPHICS Key

The [DUMP GRAPHICS] key[1] will also dump a graphics display to a printer. If a DUMP DEVICE IS statement has not been executed, the dump device is expected to be at address 701.

Aborting Graphics Dumps

If a DUMP GRAPHICS operation is aborted with the [CLR I/O] key ([Break] on an ITF keyboard), the printer may or may not terminate its graphics mode. Sending 192 null characters (ASCII code zero) to a printer such as an HP 9876 terminates its graphics mode. For example:

```
OUTPUT Dump_dev USING "#,K";RPT$(CHR$(0),192)
```

To dump a graphics image from an external color monitor which is interfaced through a 98627A at address 28, you could execute either of the following:

```
DUMP DEVICE IS Dump_dev
DUMP GRAPHICS 28
```

or

```
DUMP GRAPHICS 28 TO #Dump_dev
```

[1] On an ITF keyboard, press [Shift] with the third-from-the-left unlabeled key above the numeric keypad.

Expanded Dumps

If you want the image to be twice as large in each dimension as the actual screen size, you can execute the following two statements:

```
100   DUMP DEVICE IS 701,EXPANDED
110   DUMP GRAPHICS
```

This will cause the dumped image to be four times larger than it would be if `,EXPANDED` had not been specified. Each dot is represented by a 2×2 square of dots, and the resulting image is rotated 90° to allow more of it to fit on the page. Depending on your screen size and printer size, the image being dumped may not entirely fit on the printer. If it doesn't, it will be truncated.

Dumping Displays with Non-Square Pixels

If you have an HP 98542A or HP 98543A display, the pixels are not square. Your images are not distorted because of this, however, because the BASIC system compensates for the rectangularity. Also, when dumping graphics, the image is not distorted; again, the BASIC system compensates for the non-square pixels.

For machines which have a display with non-square pixels (such as the HP 98542A and the HP 98543A), a non-expanded DUMP GRAPHICS will produce an image that matches the CRT *only* if no alpha appears in the graphics planes. Since most printers print square pixels, this routine treats graphics pixel pairs as single elements and prints one square for each pixel pair in the frame buffer. Because alpha works with individual pixels, and not with pixel pairs, mixed alpha and graphics will appear blurred on a DUMP GRAPHICS non-expanded output. Using the EXPANDED option causes the vertical length (the height on the CRT) to be doubled as before, but dumps each separate pixel. In this mode, mixed alpha and graphics will appear the same on the dump as on the CRT.

Note that on multiplane bit-mapped displays, only graphics write-enabled planes are dumped.

Dumping to Non-Standard Printers

If you have a printer which does not conform to the HP Raster Interface Standard, all is not lost. It must, however, be capable of printing raster-image bit patterns. There are two main methods by which printers print bit-sequences.

- The first is when a printer receives a series of bits, it prints them in a one-pixel-high line across the screen. The paper then advances one pixel's distance, and the next line is printed.
- The other method (which lends itself to user-defined *characters* more than graphics image dumping) takes a series of bits, breaks it up into 8-bit chunks, and prints them as vertical bars 8 pixels high and one pixel wide. The next eight bits compose the next 1×8-pixel bar, and so forth.

Example of Dumping to an HP 82905A Printer

This latter method listed above is that used by the HP 82905A printer. The image (which is printed out sideways) takes a GSTOREd image and breaks the 16-bit integers into two 8-bit bytes, and sends them to the printer one row at a time. This is the reason for the `Hi$` and `Lo$`, the high-order (left) and low-order (right) bytes of the current integer. The following subprogram performs the function of a DUMP GRAPHICS statement from a medium-resolution display to an HP82905A printer:

```
10 Dump_graphics:    SUB Dump_graphics(OPTIONAL Dev_selector_)
20     OPTION BASE 1                         ! Arrays start at 1
30     INTEGER Y_pixels,Row,Column,Element,Char,Return_array(6)! Speed it up...
40     DIM Pad$[45]                          ! Padding for centering
50     IF NPAR=1 THEN                        ! Is output device specified?
60       Dev_selector=Dev_selector_          ! If so, use it.
70     ELSE                                  ! Otherwise,
80       Dev_selector=701                    ! Default to 701
90     END IF
100    GESCAPE CRT,3;Return_array(*)         ! Get the screen size
110    Words_per_row=Return_array(5)         ! Width of screen (in words)
120    Y_pixels=Return_array(6)              ! Height of screen (in pixels)
130    ALLOCATE Hi$[Y_pixels],Lo$[Y_pixels] ! High- and low-order bytes
140    ALLOCATE INTEGER Screen(Y_pixels,Words_per_row)  ! Screen array
150    Pad$=RPT$(CHR$(0),45)                 ! 45 nulls centers the image
160    GSTORE Screen(*)                      ! Store the picture
170    Esc$=CHR$(27)&"K"&CHR$((Y_pixels+45) MOD 256)&CHR$((Y_pixels+45) DIV 256)
180    OUTPUT Dev_selector USING "K";CHR$(27)&"A"&CHR$(8) ! 1 line=8/72 inch
190    FOR Column=1 TO Words_per_row         ! For every 16-bit swath across...
200      FOR Row=Y_pixels TO 1 STEP -1       ! and for every pixel down...
210        Element=Screen(Row,Column)        ! get appropriate array element,
220        Char=Y_pixels-Row+1               ! determine the string  subscript,
230        Hi$[Char]=CHR$(INT(Element/256)) ! fill up the high-order byte...
240        Lo$[Char]=CHR$(Element MOD 256)   ! and the low-order byte.
250      NEXT Row
```

```
260     OUTPUT Dev_selector USING "K";Esc$&Pad$&Hi$
270     OUTPUT Dev_selector USING "K";Esc$&Pad$&Lo$
280   NEXT Column
290   OUTPUT Dev_selector USING "K";CHR$(27)&"A"&CHR$(12) ! 1 line=12/72 inch
300   SUBEND
```

To DUMP GRAPHICS to other types of printers, modify the preceding subprogram appropriately for the destination device. See your printer's manual for information about its "raster-image" mode.

Negative Images

Note that on a CRT, an "on" pixel is light on an otherwise dark background, and on a printer, an "on" pixel is dark on an otherwise light background. Thus, the hard copy is a negative image of that on the screen. To dump light images on a dark background, you can invert every bit in the stored image. You can use the BINCMP function to complement the bits in every word before you send the image to the printer, or you can invert the bits of an integer by using this program segment:

```
IF N=-32768 THEN
  N=32767
ELSE
  N=-N-1
END IF
```

The reason for the subtraction is that Series 200/300 computers use two's-complement representation of integers. Also, you must consider −32 768 as a special case because you cannot negate −32 768 in an integer; +32 768 cannot be represented in a signed, sixteen-bit, two's-complement number.

Using Plotters

In Chapter 1, the program listings contained a line which said:

```
PLOTTER IS CRT,"INTERNAL"
```

This caused the computer to activate the internal CRT graphics raster as the plotting device, and thus all subsequent commands were directed to the screen.

Selecting a Plotter

If you want a plotter to be the output device, only the PLOTTER IS statement needs to be changed. If your plotter is at interface select code 7 and address 5 (the factory settings), the modified statement would be:

```
PLOTTER IS 705,"HPGL"
```

"HPGL" stands for Hewlett-Packard Graphics Language, and it is the low-level language which the plotters actually speak behind the scenes. Specifying "HPGL" tells BASIC to generate HPGL commands when it executes graphics statements, and to send them to the current plotting device. We'll take a look at HPGL in subsequent sections.

Plotter Graphics with HPGL Commands

When you are executing BASIC graphics statements and they are doing operations on a HP plotter, there is nothing preventing you from interspersing your own HPGL commands between the BASIC commands. HPGL commands can be sent to the device with OUTPUT statements; however, the preferred way is to use the GSEND statement.

```
PLOTTER IS 705,"HPGL"
FRAME
GSEND "HPGL command(s)"
MOVE X,Y
GSEND "HPGL command(s)"
DRAW X+10,Y-20
```

HPGL command sequences are terminated by a line-feed, a semicolon, or an EOI indication, which is sent by the HP-IB (Hewlett-Packard Interface Bus) END keyword. Individual commands within a sequence are typically delimited by semicolons. Note that the GSEND command sends a carriage return/line feed after the specified string.

There are many HPGL commands available, but the exact ones you will be able to use depend on the device itself. Plotters are not the only devices which use HPGL; some digitizers and graphics tablets do also. By their nature, however, they use a different subset of commands than plotters do. Following are a few of the more common or useful HPGL commands.

Example of Using HPGL: Controlling Pen Speed

If your plotter pens are getting old and tired, you probably would want to make them draw more slowly to get a better quality line. (In actuality, there are other factors which can affect line quality. For example, humidity can alter the line quality of a fiber-tipped pen.) To accomplish this, you could have a statement:

```
GSEND "VS10;"
```

"VS" stands for "*V*elocity *S*elect" and the "10" specifies centimeters per second. Thus, this statement would tell the plotter to draw at a maximum speed of ten centimeters per second. It specifies a *maximum* speed rather than an *only* speed, because on short line segments, the pen does not have time to accelerate to the specified speed before the midpoint of the line segment is reached and deceleration must begin. The range and resolution of pen speeds, and default maximum speed depend on the plotter.

Example of Using HPGL: Controlling Pen Force

On the HP 7580 and HP7585 drafting plotters, you can specify the amount of force pressing the pen tip to the drawing medium. This is useful when matching a pen type (ball-point, fiber-tip, drafting pens, etc.) to a drawing medium (paper, vellum, or mylar, etc.). Again, if a pen is partially dried out, it may help line quality to adjust the pen force.

An example statement is:

```
GSEND "FS3,6;"
```

This statement (*F*orce *S*elect) would specify that pen number 6 should be pressed onto the drawing medium with force number 3. As you can see, the force specifier occurs first, the pen number second. The reason for this is that if you do *not* specify a pen number, all pens will be affected.

The force number is translated into a force in grams. If, for example, you have an HP 7580A plotter, the force number is converted to force as follows:

1 = 10 grams	5 = 42 grams
2 = 18 grams	6 = 50 grams
3 = 26 grams	7 = 58 grams
4 = 34 grams	8 = 66 grams

Example of Using HPGL: Selecting Character Sets

Some plotters contains internal character sets which may be much more pleasing to the eye or more appropriate for your application than the character set provided by the BASIC operating system. Through HPGL, you can tell the plotter to use these character sets.

```
GSEND "CS1;"
```

tells the plotter to use *C*haracter *S*et 1 until further notice. This means, however, that to actually get these characters, you cannot use the LABEL statement in BASIC. This is because the BASIC graphics system generates all its characters as a series of line segments, and the plotter can't tell when it is told to draw a line segment whether it is going to be part of a character or not. Thus, you must use the HPGL label command, LB:

```
GSEND "LBThis is an example string."&CHR$(3)&";"
```

CHR$(3) is the End-of-Text, or ETX, control character. It is the default terminator for the LB command. If you wish, you can specify other characters to signal the end of a line of text to label. You use the *D*efine *T*erminator command:

```
GSEND "DT&;"
```

This statement instructs the plotter to consider the ampersand to be the terminator. Thus, every `LB` command must have an ampersand as the final character.

NOTE

When using a printable ASCII character as the terminator, *it will be labelled* in addition to terminating the LB command.

NOTE

There must be a terminator as the final character in the string to indicate the end of the text, or all subsequent commands will be considered text and not commands; that is, they will merely be labelled, not executed.

Error Detection when Using HPGL Commands

When using HPGL commands, there is always a possibility of making an error. When this occurs, the program should be able to respond in a friendly way, and not just hang then and there. With HPGL, it is possible to interrogate the plotting device and determine the problem. The following statements in an error-trapping routine would determine the type of error that occurred:

```
GSEND "OE;"
ENTER 705;Error
```

After these two statements have executed, the variable `Error` will contain the number of the *most recent* error. What the error code means depends on the particular device being used.

This is not by any means an exhaustive list of HPGL commands, but it serves to acquaint you with the concept of using the HPGL language, and the amount of control it gives you over the peripheral device. A thorough understanding of HPGL can only be obtained by combining information from the owner's manual of the particular device you have with actual hands-on experience.

Plotting to Files

The preceding PLOTTER IS statements in this chapter were used to direct HPGL commands to a plotter. You can also direct these commands to a file. This is useful when you want to check what is being sent to a plotter, or when you want to generate a special sequence of HPGL commands that you cannot generate with BASIC graphics statements; it is also useful when using an SRM plotter spooler, as discussed in a subsequent section.

The following statement would cause subsequent plotter output (HPGL commands) to go to a file named `Plot`.

```
CREATE BDAT "Plot:,700",20
PLOTTER IS "Plot:,700";"HPGL"
FRAME
MOVE 0,0
DRAW 100,100
MOVE 0,100
DRAW 100,0
MOVE 50,50
CSIZE 15
LABEL "Big X"
PLOTTER IS CRT,"INTERNAL"
```

Plot must be a BDAT file. Another PLOTTER IS statement, SCRATCH A, or GINIT statement closes the file. A Reset also closes the file.

Plotter Paper Sizes

The PLOTTER IS statement also provides you with the capability of specifying a non-default paper size. Here is the general syntax you can use; just substitute the limits (in millimetres) for the four parameters:

`PLOTTER IS "File","HPGL",` *Xmin, Xmax, Ymin, Ymax*

See the *BASIC Language Reference* and your plotter's manual for additional information.

Limitations

If you are performing an operation on one plotting device, and attempt to send the plot to another device which does not support that operation, it won't work.

For example: area fills, which are valid operations on most displays, are not available on plotters. Color map operations, which are valid on displays, are not valid on a plotter. Erasing lines can be done on displays, but, naturally, not on a hard-copy plotter. On the other hand, HPGL commands will be interpreted correctly by a hard-copy plotter, but not by a display.

Using GSEND with PLOTTER IS Files

This statement sends a string of characters to the current PLOTTER IS device, which may be a file or a plotter.

```
GSEND "LBThis is an example HPGL command string."&CHR$(3)&";"
```

The string is to contain Hewlett-Packard Graphics Language (HPGL) command(s). GSEND is useful when the PLOTTER IS device is a file, since it is not possible to OUTPUT an HPGL command to the file while it is the PLOTTER IS device.

```
CREATE BDAT "Plot:,700",20
PLOTTER IS "Plot:,700";"HPGL"
FRAME
MOVE 0,0
DRAW 100,100
MOVE 0,100
DRAW 100,0
MOVE 20,20
GSEND "CS2;"
GSEND "LBThis is X in the plotter's character set 2."&CHR$(3)&";"
PLOTTER IS CRT,"INTERNAL"
```

Note that HPGL syntax is **not** checked by the BASIC system. Therefore, you will need to take extra care when using HPGL commands in this manner.

Using SRM Plotter Spoolers

The SRM system not only provides shared access to plotters, but also manages their use so that workstations never need to wait for output to be generated.

What Are Spoolers?

To use shared plotters, you place files to be output into a special directory where they are held until the plotter is free. This method is called "spooling," and the directory where the files are kept is called the "spooler directory." After a file is placed in a spooler directory, the workstation is free to do other processing.

Setting Up a Plotter Spooler

Spooler directories are created for the SRM server's use when the shared peripherals are installed on the SRM system. Setting up a spooler directory is explained in the "Interfaces and Peripherals" chapter of the *SRM System Manager's Guide*. The examples in this section assume that a spooler directory named PL (for "PLotter") has already been created in the SRM root directory.

Preparing Plotters

If your plotter does not feed its paper automatically, a message appears on the SRM server's screen indicating that the plotter needs to be set up. After you have put paper on the plotter, you may begin the plotting by using the server's SPOOL CONTINUE command (described in the *SRM System Manager's Guide*). Plotters with automatic paper feed require no operator intervention.

Plotter Spooling

These are the steps in using the SRM plotter spooler:

1. Create a file.
2. Specify it as the PLOTTER IS device.
3. Perform BASIC or HPGL plotting operations.
4. When finished, close the file and if it is not already there COPY or RENAME it into the spooler directory.
5. Wait for the file to be output to the plotter.

Example of Plotting to a File

You can use the PLOTTER IS statement to send data to a file, which you can later send to an SRM plotter. The following command sequence illustrates this spooling method:

```
CREATE BDAT "/PL/Plot_file",1
PLOTTER IS "/PL/Plot_file"
FRAME
MOVE 0,0
DRAW 100,100
MOVE 0,100
DRAW 100,0
GSEND "CS2;"
GSEND "LBThis is X in the plotter's character set 2."&CHR$(3)&";"
PLOTTER IS CRT,"INTERNAL"
```

PLOTTER IS works only with BDAT files. Because the SRM 1.0 operating system's spooling works only with ASCII files, you cannot use PLOTTER IS for plotter spooling with that version of SRM.

Note

The DUMP DEVICE IS statement does **not** support directing data to files as shown above, so it cannot be used for plotter spooling.

Checking the Spooler's Status

The SRM spooler waits until the file is non-empty and closed before sending its contents to the output device. If your file is not plotted within a reasonable amount of time, you may not have closed it. You can verify that your file is ready to be plotted by cataloging the spooler directory:

`CAT "/PL"` [Return] or [ENTER]

The open status (`OPEN STAT`) of the file currently being printed or plotted is listed as locked (`LOCK`). Files currently being written to the spooler directory are listed as `OPEN`. Files that do not have a status word in the catalog are ready for plotting.

Version 2.0 (and later versions) of the SRM operating system allow BDAT files to be sent to the printing device as a byte stream. (With SRM version 1.0, only ASCII files can be used.)

Aborting Plotting In Progress

To abort an in-progress plotting, use the SPOOL ABORT command from the SRM server. The system stops sending data to the output device and closes, then purges the file. For details on bringing the spooler UP and DOWN, see the description of the SPOOLER command in the "Language Reference" section of the *SRM System Manager's Guide.*

With the SRM 2.0 operating system, if a plotter is taken off-line while a file is being spooled, the spooler stops and resumes when the plotter is put back on-line. No data is lost during such an interruption. The SRM 1.0 operating system also resumes plotting, but from the beginning of the file.

Dumping Graphics to a Printer Spooler

This program shows an example of using GSTORE to capture the pixels on a graphics raster, which is then sent to an SRM printer spooler file.

```
100  ! Program to DUMP GRAPHICS to an SRM plotter spooler.
110  ! Works with 2608A/S, 2631G, 293x, 2563A, and other printers.
120  !
130  ! Parameters will vary for different graphics raster sizes.
140  !   Picture(*) is:  300x25  for 9816 & 9826 (300*400 pixels)
150  !                   390x32  for 9836        (390*512 pixels)
160  !
170  !   Buffer_(*) is:  1:8704  for 9816 & 9826
180  !                   1:14044 for 9836
190  !
200  !   Rowlength  =    25 for 9816 & 9826
210  !                   32 for 9836
```

```
220  !
230  !   Rows       =    300 for 9816 & 9826
240  !                  390 for 9836
250  !
260  INTEGER Picture(1:300,1:25),Buffer_(1:8704) ! For 9816 & 9826.
270  INTEGER Bufptr,Word_,Row_,Rows,Rowlength,Byte1,Byte2,Temp
280  Rowlength=25
290  Rows=300
300  !
310      .
        . Draw picture on display raster (using BASIC graphics).
         .
400  GSTORE Picture(*)
410                     ! Generate "escape" sequences for graphics mode.
420  Buffer_(1)=6954     ! "Esc*"
430  Buffer_(2)=29249    ! "rA"
440  Bufptr=3
450  Temp=Rowlength+Rowlength
460  Byte1=NUM("b")*256+((Temp DIV 10)+48)
470  Byte2=((Temp MOD 10)+48)*256+NUM("W")
480  !
490  FOR Row_=1 TO Rows
500                            ! Identify number of bytes for this line.
510    Buffer_(Bufptr)=6954     ! "Esc*"
520    Buffer_(Bufptr+1)=Byte1  ! "bn"
530    Buffer_(Bufptr+2)=Byte2  ! "nW"
540    Bufptr=Bufptr+3
550    FOR Word_=1 TO Rowlength
560      Buffer_(Bufptr)=Picture(Row_,Word_)
570      Bufptr=Bufptr+1
580    NEXT Word_
590  NEXT Row_
600                      ! Terminate graphics mode.
610  Buffer_(Bufptr)=6954 ! "Esc*"
620  Buffer(Bufptr+1)=29250 ! "rB"
630  !
640  ! Store Picture in spooler directory, so that it will be printed.
650  CREATE BDAT "/LP/PICTURE",1
660  ASSIGN @Dump TO "/LP/PICTURE"
670  OUTPUT @Dump;Buffer_(*)
680  ASSIGN @Dump TO *
690  !
700  END
```

Interactive Graphics and Graphics Input 4

Interactive Graphics and Graphics Input

4

Introduction

It has already been pointed out that graphics is a very powerful tool for communication. The high speed of graphics operations on Series 200/300 computer makes possible a powerful mechanism for communicating with the computer: *Interactive Graphics.*

An Example

One way to understand interactive graphics is to see it in action. If your computer has a knob or mouse, LOAD and RUN the program "BAR_KNOB", from your *Manual Examples* disc.

If you turn the knob clockwise, the bar graph displayed on the screen will indicate a larger value. At the same time, the numeric readout underneath the bar will increase it's value. Turning the knob counterclockwise has the opposite effect. This is an effective demonstration of all the key characteristics of an interactive graphics system.

Elements of an Interactive Graphics System

The elements of this type of graphics system are as follows:

- A graphic display that represents the contents of the data structure.
- An input mechanism for interacting with the displayed image (the knob or mouse, in this case.)
- A data structure. (The value displayed underneath the bar is the contents of a variable that we are modifying. The BASIC variable containing the value is considered a degenerate case of a data structure.)

This is the minimum set of requirements for an interactive graphics system. A key feature of interactive graphics is that it is a *closed loop* system. This means that the operator can immediately see the effect of his action on the system, and thus base his next action not only on the state of the system, but also on the effect his last action had on the system. A few points are worth noting about this system:

- The knob or mouse is used because it is *functionally appropriate.* While we could have used softkeys, a tablet or mouse, or entered numeric values to control the bar graph, the knob "feels" right. We are used to using knobs to control bar graph metered readouts.

- Control of the value with the knob is fairly intuitive. The normal range markings make it readily apparent when the value is in range. Little explanation is needed, due to the immediate feedback from the displayed image.
- A system is "modeled." The user's input must have a well defined relation to the output of the system. This relation, or rule, should be easy to learn (via experimentation and immediate feedback).

Thus, interactive graphics can be as simple as representing a single value on the screen and providing the user a method for interacting with it. It can also be as complex as a Printed Circuit layout system. This chapter will not tell you how to build a Printed Circuit layout system, but it will provide some hints on implementing interactive graphics systems that work.

Characterizing Graphic Interactivity

One of the most important things in designing a good interactive graphics system is characterizing the interaction with the system correctly. Properly characterizing the interactivity allows selection of the most appropriate device for interaction with the system. Three things have to be considered in characterizing the interaction:

- The number of *degrees of freedom* in the system. This is the number of ways in which a system can be changed.
- The *quality* of each of the degrees of freedom. This describes how the input to a degree of freedom can be changed.
- The *separability* of the degrees of freedom.

The BAR_KNOB program has limitations in these regards. Read on to see why.

Selecting Input Devices

The purpose of the discussion on characterization of graphic interaction was to lay the groundwork for discussing when various input devices are appropriate. There are several available computers, and choosing the correct one is critical to the design of a highly productive human interface for an interactive graphics program.

Single Degree of Freedom

Many interactive graphics programs need deal only with a single degree of freedom. The appropriate control device for such programs depends on whether continuous control or quantizable control is needed.

The program "BAR_KNOB" is a good example of a single degree of freedom that is continuous. The knob is ideal for controlling a program like this. If "fine tuning" is needed, the shift key can be used as a multiplier to change the interpretation of the knob. It is also possible to use the softkeys for fine tuning.

Softkeys can be used for quantizable control of a degree of freedom. It is also possible to use keyboard entry of numeric values for quantizable information. Remember that sofkey labels range from [f1] through [f8] on some keyboards, and from [k0] through [k9] on others. Also, if you have an ITF keyboard which provides different menus for the System menu and User 1 through 3 menus, then the SYSTEM KEYS and USER *n* KEYS statements enable you to switch from one menu to another. For example, `USER 1 KEYS` displays the menu for User 1 softkeys, and `SYSTEM KEYS` displays the System softkeys menu. Use a 2 or 3 for User 2 or User 3 menus.

Non-Separable Degrees of Freedom

One characteristic of multiple, non-separable degrees of freedom is that they are generally continuous. The most common operation of this type is free-hand drawing. This is most easily accomplished with a graphics tablet.

Separable Degrees Of Freedom

In many programs, the degrees of freedom are completely separable. In fact, for some operations, it is definitely preferable to have totally independent control of the degrees of freedom of the model.

All Continuous

If all the degrees are continuous, a good choice is using the softkeys to select the degree of freedom and then using the knob to control the input to that degree of freedom. An even better choice is to use an HP 46085 Control Dial Box, which has nine knobs. See the "CDials" program on the *Manual Examples* disc.

All Quantizable

If all the degrees are quantizable, using softkeys is ideal.

Mixed Modes

In most sophisticated graphics systems, several degrees of freedom in the system interact with each other. A good example is a graphics editor. In a graphics editor, your primary interaction is with a visual image, and the degrees of freedom (X and Y location) for that operation are partially separable, at best. (They are non-separable if it supports freehand drawing.) There is also a degree of freedom involved in controlling the program. The program control is strongly separable from the image creation operation.

The most appropriate device for supporting mixed modes is a graphics tablet. The HP 9111A tablet supports two modes of interaction by partitioning the digitizing surface into two areas. Sixteen small squares along the top of the tablet are used as softkeys to provide a control menu. The large, framed area underneath the softkeys is the active digitizing area. The active digitizing area is used for interacting with the image you are creating. Some HP-HIL tablets (such as the HP 46087A and HP 46088A) use a 4-button stylus, or "puck," which has physical buttons on the cursor device.

It is possible to combine the quantized, separable control operations with continuous, non-separable image editing. This is done by using the active digitizing area for interacting with the image and using the menu area for controlling the operations available in the editing program. The operator does not have to change control devices to access the different interaction modes.

Echoes

An important part of interactive graphics is letting the operator know "where he is at." This can be done by updating the image (as in "BAR_KNOB".) In other operations—such as menu selection, object positioning, and freehand drawing—it is important to show the operator where he is. In many cases, this can be done with the SET ECHO statement.

The Built In Echo

Many graphics applications can be handled using the built in echo. Executing TRACK...IS ON sets up the system to track the graphics input device with the built in echo during a DIGITIZE. The following program shows how to do single-point digitizing with the built in echo.

```
100      GINIT                                   ! Restore defaults
110      GRAPHICS INPUT IS 706,"HPGL"            ! 9111 is input
120      PLOTTER IS CRT,"INTERNAL"               ! (Redundant)
130      TRACK CRT IS ON                         ! Enable tracking
140      !
150      GRAPHICS ON
160      VIEWPORT 0,133,0,100                    ! Match aspect ratios
170      WINDOW .50,50,.20,20                    ! Define GDUs
180      FRAME                                   ! Draw bounds
190      AXES 10,10,0,0,5,5                      ! Draw axes
200      MOVE 0,0                                ! Begin at origin
210      !
220      !
230 Track:       DIGITIZE X,Y,Status$            ! Request coords
240          ! updating cursor until coords received
250          !
260          DRAW X,Y                        ! Connect points
270             !
280    GOTO Track
290      !
300      END
```

The TRACK..IS ON statement merely enables the tracking feature; the actual tracking is performed while the DIGITIZE statement is being executed. The locator is "tracked" by moving the output device's "cross-hair" (or pen) correspondingly. Notice that the definition of the DIGITIZE statement has been modified slightly—now its execution causes the locator to be tracked and "echoed" on the output device until the stylus (or Digitize button) is pressed.

After the stylus is pressed, the DIGITIZE statement has finished execution and the DRAW statement is executed. This program draws lines between the digitized points, but you may want to change this response as desired with the appropriate software.

If accuracy is not exceptionally important, you can do continuous digitizing with the READ LOCATOR statement.

This program, as it stands, will only work with an HP 9111 graphics tablet at address 706. If you wish to use a mouse or an HP-HIL tablet, change the GRAPHICS INPUT IS statement to :

`GRAPHICS INPUT IS KBD,"KBD"` *(for HP-HIL Mouse)*
or
`GRAPHICS INPUT IS KBD,"TABLET"` *(for HP-HIL Tablet)*

The following program continuously tracks the input locator and monitors the pressed/not-pressed status of the Digitize button (or stylus). The cursor position is continuously echoed on the output device, and lines are drawn if the Digitize button (or stylus point) is pressed.

```
100      GINIT                                    ! Restore defaults
110      GRAPHICS INPUT IS 706,"HPGL"             ! Define input
120      PLOTTER IS CRT,"INTERNAL"                ! Define output
130      GRAPHICS ON
140      VIEWPORT 0,133,0,100                     ! Match aspect ratios
150      WINDOW 0,100,1,100                  ! Define UDUs
160      FRAME                                    ! Draw limits
170      !
180      LOOP
190        READ LOCATOR X,Y,Status$
200        SET ECHO X,Y
210        Button$=Status$[1,1]
220        GOSUB Action
230      END LOOP
240      !
250 Action:    IF Button$="0" THEN MOVE X,Y
260             IF Button$="1" THEN DRAW X,Y
270             RETURN
280            !
290      END
```

Making Your Own Echoes

In some applications, the cross-hair generated by SET ECHO is not sufficient. You may want to generate a *rubber band* line or box. A rubber band line is stretched from an anchor point to the echo position. In these cases, it is necessary to draw your own echo.

Since an echo needs to be repositioned as the operator interacts with it, it must be constantly drawn and redrawn. If it is just drawn and then erased, the background it is drawn over will soon become littered with erased images of the echo. What we really want to do is find a way to draw it and then "undraw" it, rather than erasing it. The complementary drawing mode is used to do this. In the complimentary drawing mode, the bits specified by the current pen selector are complimented in the frame buffer, rather than just overwriting the contents. If a second complimenting is done, the image is restored to whatever was there before the echo was written to it. The echo generated by SET ECHO is automatically drawn in the complimentary mode.

It is important to remove any echo you have drawn on the screen **before** updating the image. The complimenting of a bit pattern does not restore the image if the image was altered between the complimentary drawing and undrawing. This is done automatically by SET ECHO, but you must handle it yourself if you are building your own echoes. The following loop will support a tablet with several different echoes, when used with the echo routines discussed below.

```
570   LOOP                                       ! Main Tracking Loop
580     READ LOCATOR Xin,Yin
590     DISABLE
600     CALL Make_echo(Xin,Yin,Echo_type)        ! Several Echo Types
610     ENABLE
620   END LOOP
```

Two sets of echo routines are provided, one set for monochrome and one set for color systems. Both a `Kill_echo` and a `Set_echo` routine are provided for each case.

Monochrome Echoes

The complementary drawing mode can be accessed for making your own echo by selecting PEN 0. The subroutines which follow are used to implement rubber band line and rubber band box echoes. Be aware that the subroutines would be a part of some greater program that you create. The intent is to demonstrate techniques.

```
2920 Kill_echo:SUB Kill_echo
2930    !************************************************************
2940    !*                                                          *
2950    !*  This routine gets rid of whatever echo is left over on the *
2960    !*  screen.                                                 *
2970    !*                                                          *
2980    !************************************************************
2990    !
3000    COM /Echo_local/ Last_x,Last_y,Last_anchor_x,Last_anchor_y
3010    COM /Echo_local2/ Last_pen,Last_echo_type
3020    COM /Echo_global/ Echo_drawn,Anchor_x,Anchor_y
3030    COM /Booleans/ INTEGER True,False
3040    COM /Modals/ INTEGER Drawmode,Normal,Complement,Current_pen,
Current_fill
3050    COM /Echo_global1/ Rubber_line,Cross,Rubber_box
3060    !
3070    PEN 0
3080    SELECT Last_echo_type
3090    CASE Rubber_line
3100      MOVE Last_anchor_x,Last_anchor_y
3110      DRAW Last_x,Last_y
3120    CASE Rubber_box
3130      MOVE Last_anchor_x,Last_anchor_y
3140      RECTANGLE Last_x-Last_anchor_x,Last_y-Last_anchor_y
3150    CASE ELSE
3160    END SELECT
3170    Echo_drawn=False
3180    PEN 1
3190  SUBEND
3200    !
3210 Make_echo:SUB Make_echo(X,Y,Echo_type)
3220    !******************************************************************
3230    !*                                                                *
3240    !*  This routine makes the an echo of the current Echo_type at the *
3250    !*  specified (X,Y) location.  It also updates the variables for  *
3260    !*  the Kill_Echo Subprogram.                                     *
3270    !*                                                                *
3280    !******************************************************************
3290    !
3300    COM /Echo_local/ Last_x,Last_y,Last_anchor_x,Last_anchor_y
3310    COM /Echo_local2/ Last_pen,Last_echo_type
3320    COM /Echo_global/ Echo_drawn,Anchor_x,Anchor_y
3330    COM /Booleans/ INTEGER True,False
```

```
3340    COM /Modals/ INTEGER Drawmode,Normal,Complement,Current_pen,
Current_fill
3350    COM /Echo_global1/ Rubber_line,Cross,Rubber_box
3360    COM /Bounds/ Max_clip_y
3370    !
3380    IF Echo_drawn THEN CALL Kill_echo
3390    IF Y<Max_clip_y THEN
3400      PEN 0
3410      SELECT Echo_type
3420      CASE Rubber_line
3430        MOVE Anchor_x,Anchor_y
3440        DRAW X,Y
3450      CASE Rubber_box
3460        MOVE Anchor_x,Anchor_y
3470        RECTANGLE X-Anchor_x,Y-Anchor_y
3480      CASE ELSE
3490      END SELECT
3500      SET ECHO X,Y
3510      Last_x=X
3520      Last_y=Y
3530      Echo_drawn=True
3540    END IF
3550    SET ECHO X,Y
3560    Last_echo_type=Echo_type
3570    Last_anchor_x=Anchor_x
3580    Last_anchor_y=Anchor_y
3590    PEN 1
3600  SUBEND
```

Color Echoes

Accessing the complementary drawing mode is slightly different in color. The complimentary drawing mode can be accessed for making your own echo by specifying a negative pen number after a GESCAPE to select the non-dominant writing mode (operation selector of 5). The subroutines are used to implement rubber band lines, and have hooks in place for rubber band boxes. **`Complement`** has been initialized to 5, and **`Drawmode`** contains the current drawing mode. Again, remember to study the subroutine listing to examine programming techniques.

```
9900 Kill_echo:SUB Kill_echo
9910    !*****************************************************************
9920    !*                                                               *
9930    !*  This routine gets rid of whatever echo is left over on the   *
9940    !*  screen.                                                      *
9950    !*                                                               *
9960    !*****************************************************************
9970    !
9980    COM /Echo_local/ Last_x,Last_y,Last_anchor_x,Last_anchor_y
9990    COM /Echo_local2/ Last_pen,Last_echo_type
10000   COM /Echo_global/ Echo_drawn,Anchor_x,Anchor_y
10010   COM /Booleans/ INTEGER True,False
10020   COM /Modals/ INTEGER Drawmode,Normal,Complement,Current_pen,
Current_fill
10030   COM /Echo_global1/ Rubber_line,Cross,Rubber_box
10040   !
10050   GESCAPE 3,Complement
10060   IF Last_pen<>0 THEN
10070     PEN -Last_pen
10080   ELSE
10090     PEN -1
10100   END IF
10110   SELECT Last_echo_type
10120   CASE Rubber_line
10130     MOVE Last_anchor_x,Last_anchor_y
10140     DRAW Last_x,Last_y
10150   CASE ELSE
10160   END SELECT
10170   GESCAPE 3,Drawmode
10180   PEN Current_pen
10190   Echo_drawn=False
10200 SUBEND
10210   !
10220 Make_echo:SUB Make_echo(X,Y,Echo_type)
10230   !*******************************************************************
10240   !*                                                                 *
10250   !*  This routine makes the an echo of the current Echo_type at the *
10260   !*  specified (X,Y) location.  It also updates the variables for   *
10270   !*  the Kill_Echo Subprogram.                                      *
10280   !*                                                                 *
```

```
10290   !******************************************************************
10300   !
10310   COM /Echo_local/ Last_x,Last_y,Last_anchor_x,Last_anchor_y
10320   COM /Echo_local2/ Last_pen,Last_echo_type
10330   COM /Echo_global/ Echo_drawn,Anchor_x,Anchor_y
10340   COM /Booleans/ INTEGER True,False
10350   COM /Modals/ INTEGER Drawmode,Normal,Complement,Current_pen,
Current_fill
10360   COM /Echo_global1/ Rubber_line,Cross,Rubber_box
10370   COM /Bounds/ Max_clip_y
10380   !
10390   IF Echo_drawn THEN CALL Kill_echo
10400   IF Y<Max_clip_y THEN
10410     GESCAPE 3,Complement
10420     IF Current_pen<>0 THEN
10430       PEN -Current_pen
10440     ELSE
10450       PEN -1
10460     END IF
10470     SELECT Echo_type
10480     CASE Rubber_line
10490       MOVE Anchor_x,Anchor_y
10500       DRAW X,Y
10510       Echo_drawn=True
10520     CASE Cross
10530     CASE ELSE
10540     END SELECT
10550     GESCAPE 3,Drawmode
10560     SET ECHO X,Y
10570     Last_x=X
10580     Last_y=Y
10590   END IF
10600   SET ECHO X,Y
10610   Last_echo_type=Echo_type
10620   Last_anchor_x=Anchor_x
10630   Last_anchor_y=Anchor_y
10640   Last_pen=Current_pen
10650   PEN Current_pen
10660 SUBEND
```

Graphics Input

In many interactive graphics applications the tablet is used as an echo mover. The transformation between the graphics tablet and the display should be linear in such applications, but the axes do not have to transform through the same scaling. It doesn't matter if a square on the tablet represents a square on the display if you are just using the tablet to move a cross-hair on the display. However, if you are trying to copy an image from paper image to the display (using a graphics tablet) it is important to preserve both the linearity and the aspect ratio in the transformations.

The *maximum usable area* of a graphics device is bounded by its *hard clip limits*; for example, the pen cannot be made to draw outside these limits on an output device.

The *current usable area* is bounded by the rectangle defined by the points P1 and P2; the lower-left corner is P1, and the upper-right corner is P2. On many devices, these points can be moved manually or by the program.

When the GINIT statement or a PLOTTER IS statement is executed, points P1 and P2 are read from the plotting device; with GINIT, the plotting device is assumed to be the internal CRT. The value of RATIO is then set to the result of the following calculation:

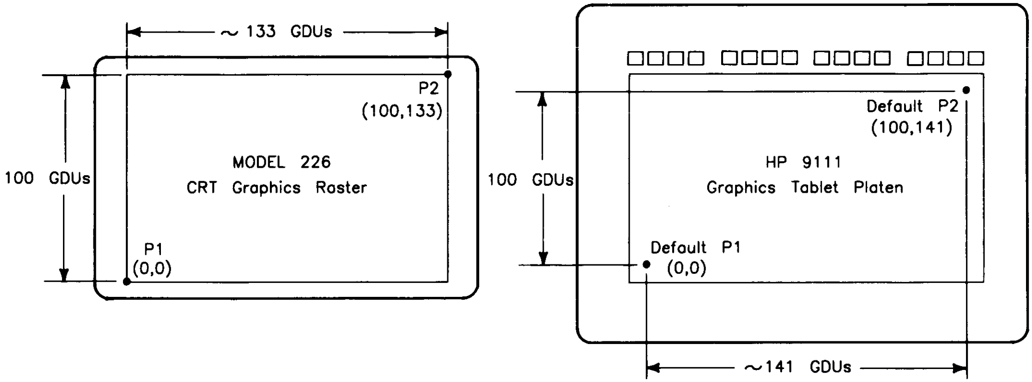
RATIO=(P2x−P1x)/(P2y−P1y)

GINIT does a WINDOW for the internal CRT only; PLOTTER IS does not do implicit windowing. You must explicitly do the following statements:

If RATIO >= 1:
```
VIEWPORT 0,100*RATIO,0,100
WINDOW 0,100*RATIO,0,100
```

If RATIO < 1:
```
VIEWPORT 0,100,0,100/RATIO
WINDOW 0,100,0,100/RATIO
```

As seen above, the X and Y coordinates of P1 are always both 0 Graphic Display Units. The default Graphic Display Unit coordinates of P2 depend on the device; however, the smaller coordinate of this point is always 100 Graphic Display Units. Two examples are shown below:



Usable-Area Boundaries:

Left edge = X coordinate of P1	Right edge = X coordinate of P2
Bottom edge = Y coordinate of P1	Top edge = Y coordinate of P2

Figure 4-1. Default Locations of P1 and P2 Using a Model 226 and HP 9111

When a PLOTTER IS statement is executed, the locations of points P2 and P2 on the specified device are determined. The current VIEWPORT statement parameters are then used to define the physical area (in GDUs) which is to be scaled (in UDUs) by the WINDOW or SHOW statement currently in effect.

When a subsequent GRAPHICS INPUT IS statement is executed, the operating system attempts to apply the current VIEWPORT and WINDOW (or SHOW) parameters to the P1,P2 rectangle of the input device. In the preceding example, the two usable areas are not identical in size (in GDUs), since the HP 9111 has a smaller horizontal-to-vertical aspect ratio. This difference in aspect ratios may produce **two types of potentially undesirable results** when using these two devices together for interactive graphics capabilities. The GRAPHICS INPUT IS sets the hard clip limits of the input device to the largest space possible that has the same aspect ratio as the output device.

HP-HIL Devices

The HP-HIL family of peripherals is a relatively new set of devices which communicate via HP-HIL (Hewlett-Packard Human Interface Loop). The HP-HIL graphics input devices include the knob, mouse, and graphics tablets. The KBD binary is required for all HP-HIL devices except keyboards.

HP-HIL Relative Locators (Such as Cursor Keys, Knob, or Mouse)

The term "relative locator," in the context of a graphics input device, refers to a device which returns incremental X,Y offsets. The computer uses these incremental values to update the logical coordinates of the graphics locator. No fixed physical reference exists.

Relative locators can also provide "keystrokes" to represent the various buttons, but this capability is not available while the relative locator is being used for graphics input.

The button which is pressed is reported to the main program by the seventh and eighth bytes of the status string returned by DIGITIZE and READ LOCATOR. To determine if any buttons were pressed, check VAL(Status$). If the value is non-zero, a button was pressed. To determine *which* button was pressed, check the appropriate bit in the number represented by the ASCII characters in positions seven and eight of the string. For example:

```
ALLOCATE Button(0:Max_buttons)
      :
FOR Bit=0 TO Max_buttons
  Button(Bit)=BIT(VAL(Status$[7]),Bit)
NEXT Bit
      :
IF Button(⟨n⟩) THEN . . .
```

The keyboard is a relative device, also. For example, pressing Enter will trigger a DIGITIZE with a status string of "1,2,0,00". However, this does not show up in a READ LOCATOR operation.

Executing GRAPHICS INPUT IS KBD,"KBD" turns on *all* relative locators and the results are combined. Thus, you can safely intermix input from arrow keys, an internal knob, external knobs, and mice.

HP-HIL Absolute Locators (Such as a Tablet or TouchScreen)

An "absolute locator" is a locator which has a finite mapping area, such as the HP 9111 tablet or the 35723A Touchscreen. On these devices, the data returned are X,Y coordinate pairs. Each possible value of these pairs corresponds to a fixed location on the physical surface of the digitizing device.

"Proximity," in the context of the following paragraphs, is defined as the area in which a locator can detect that you are pointing to something. For example, on a 35723A Touchscreen, proximity is where your finger is close enough to the screen that the computer can assign a location to your finger's "shadow." For a graphics tablet, proximity is where the tablet can detect the presence of the stylus or puck.

The Touchscreen is supported as a `GRAPHICS INPUT` device—a low-resolution `TABLET`. Going into proximity on a Touchscreen—pointing to something, with your finger touching the screen—causes the Touchscreen to sense your finger's location. Going out of proximity on a Touchscreen—removing your finger from the screen—triggers a `DIGITIZE`. On the Touchscreen and the 45911A, 46087A and 46088A tablets, when out of proximity, bytes 7 and 8 of the "device status" string contain the ASCII characters "`64`".

The HP-HIL tablets consider all buttons unpressed when out of proximity. Buttons which are held down while moving into proximity are sent as key presses at the proximity transition. This can trigger a digitize at that point.

Similar to the combining of relative locators, executing `GRAPHICS INPUT IS KBD,"TABLET"` turns on all absolute locators and combines the results. However, since the display can be scaled to only one absolute locator at a time, and since the inputs replace each other (as opposed to adding to each other), this is not a useful feature.

For an program that supports the use of all of these devices, LOAD and RUN the "KBD_ICONS" program on the *Manual Examples* disc.

The `GESCAPE` statement, in conjunction with the device selector `KBD`, allows you to both set and read hard clip limits for absolute locators on the HP-HIL bus. Note that the hard clip limits are only the right-most and uppermost limits; the left and bottom edges of the plotting surface are always zero. For example, to set the hard clip limits for an HP-HIL Touchscreen in spite of the presence of a tablet on the bus:

```
10  INTEGER Parameter_array(1:2)
20  Parameter_array(1)=52
30  Parameter_array(2)=46
40  GESCAPE KBD,20,Parameter_array(*) ! Set absolute-locator hard limits
```

To read the hard clip limits for all the absolute locators on the HP-HIL bus, you can use `GESCAPE` with operation selector 21 or 22. For example:

```
10   INTEGER Param_array(1:4)  ! We have TWO absolute locators on the HP-HIL
20   GESCAPE KBD,22;Param_array(*)  ! Read ALL absolute locator limits
```

If `Param_array` had been larger that four elements, the first unused element—after using two elements for each absolute locator on the loop—would contain −1. This special value is to indicate that there are no more coordinate pairs.

Unlike other `GESCAPE` operation selectors, operation selectors 20 through 22 do not require the device at the specified select code to be currently active. Indeed, if you want to set the hard clip limits, `GESCAPE KBD,20` must be executed *before* the `GRAPHICS INPUT IS KBD,"TABLET"`.

Operation selectors 20 and 21 will give `DEVICE NOT PRESENT` errors if no tablet, Touchscreen, or HP-HIL interface exists. An operation selector 22, under the same circumstances, will return a −1 for the first entry in the return array.

The HP-HIL Tablets can be treated as a superset of the HP 9111A when used with the built-in GRAPHICS INPUT IS, READ LOCATOR, and DIGITIZE commands. The HP-HIL Tablets are a superset because of the extra button information available in the READ LOCATOR and DIGITIZE status strings.

There is one important difference between the HP 9111A and the HP-HIL Tablets which may cause programs that work with the HP 9111A to have problems with the HP-HIL Tablets. When a READ LOCATOR or DIGITIZE command is executed for an HP 9111A, the system sends a request to the HP 9111A for the current location of the stylus. The HP-HIL Tablets, on the other hand, send the current location only when it changes. They cannot be asked for the current location. When `GRAPHICS INPUT IS KBD,"TABLET"` is executed, the internal variables representing the state of the HP-HIL Tablet are initialized. Since BASIC does not know the true state of the tablet at the time, the initialization sets up a unique state which can become valid with any set of received data for the tablet, but which can also be easily recognized as invalid. This state is in proximity with negative device coordinates.

The difference between the HP 9111A and the HP-HIL Tablets should not be a problem for a program which executes the GRAPHICS INPUT IS statement initially and leaves graphics input active while it runs. However, it may be a problem if the program was written to do a GRAPHICS INPUT IS before each call to READ LOCATOR because this causes initilization to occur just before input. Note that after `GRAPHICS INPUT IS KBD,"TABLET"` is executed, the data returned by READ LOCATOR will be recognizably invalid until a transaction on the tablet causes valid data to be sent to the controller.

Support of HP-HIL Devices for Graphics Input

Here is a generic description of the HP-HIL devices for which BASIC provides drivers[1]:

- Relative locators with 1 to 3 dimensions and with up to 6 buttons;
- Absolute locators with 1 or 2 dimensions and with up to 6 buttons and/or proximity.

For both of these categories of devices, extra dimensions or multiple sets of axes disqualify the device. Note that the maximum number of buttons is six; any "seventh" button would be ignored.

Note

BASIC only configures the HP-HIL bus at power-up and **SCRATCH A**. Reconfiguring the bus physically without doing a **SCRATCH A** can result in devices not being recognized and/or in data being misinterpreted as coming from another type of device.

---

1 You can also write your own drivers for other devices. See the "HP-HIL Interface" chapter of *BASIC Interfacing Techniques*.

Dealing With Multiple Buttons

One feature of the 46060A mouse and the 46089A digitizer puck is that there are multiple buttons available to press when digitizing. This means that there is a choice, when digitizing, of how to signal the computer that you've made your choice.

The way that the computer finds out which button you pressed is by the status string returned from `DIGITIZE` and `READ LOCATOR`. The following example program merely tracks, on the CRT, the stylus movements on the digitizer. Note the use of this status string in the following example program.

```
10    ! Program "Multibutn"
20    GINIT
30    PLOTTER IS CRT,"INTERNAL"
40    VIEWPORT 0,100*RATIO,0,100
50    WINDOW 0,100,0,100
60    TRACK CRT IS ON
70    GRAPHICS INPUT IS KBD,"TABLET"
80    Status$="0,0,0,00"
90    REPEAT
100     REPEAT
110       Old_status$=Status$
120       READ LOCATOR X,Y,Status$
130       SET ECHO X,Y
140       Buttons=VAL(Status$[7,8])
150     UNTIL Buttons-BINAND(Buttons,VAL(Old_status$[7]))! Note button
presses,
160     IF Buttons=64 THEN                               ! ignore releases
170       PRINT "You exited proximity."
180     ELSE
190       PRINT USING "#,K";"You pressed "
200       Num_buttons=0                                         ! \  Count
210       FOR Button=1 TO 6                                     !  \ the
220         IF BIT(Buttons,Button-1) THEN Num_buttons=Num_buttons+1!  /   buttons
230       NEXT Button                                           ! /  pressed
240       IF Num_buttons=0 THEN PRINT USING "#,K";"no "
250       PRINT USING "#,K";"button"
260       IF Num_buttons<>1 THEN PRINT USING "#,K";"s"
270       FOR Button=1 TO 6
280         IF BIT(Buttons,Button-1) THEN
290           PRINT USING "#,K";" ",Button
300           Num_buttons=Num_buttons-1
310           SELECT Num_buttons
320           CASE 0
330             ! Do nothing
340           CASE 1
350             PRINT USING "#,K";" and"
360           CASE 2 TO 6
370             PRINT USING "#,K";","
```

```
380           END SELECT
390         END IF
400       NEXT Button
410       PRINT "."
420     END IF
430   UNTIL Buttons=11 ! Stop on simultaneous buttons 1, 2, and 4
440   GINIT            ! Get rid of the cursor
450   PRINT "Program finished."
460   END
```

Menu-Picking

Perhaps one of the most common uses for the Touchscreen is that of presenting several options on the screen, and having the user select one by pointing to it. The following example program does just that. It presents five options on the screen, and the user picks one. All the main program does after that is state which option the user picked; you may enhance and modify the example to fit your application.

Points of note in the example:

- The function `FNMenu` is a general-purpose function to which a menu array can be passed, and from which the selected option is returned.
- The user must indicate a location *within* the option boxes; if not, a warning will be given, and the user must try again. This prevents invalid data from being returned to the calling routine.
- The options are `LABEL`ed in graphics, rather than being `PRINT`ed in alpha. This avoids the difficulty in aligning alpha text and graphics scaling, which is where the `DIGITIZE` works from.

```
10    ! Program "Pick"
20    OPTION BASE 1
30    DIM Menu$(5)[20]
40    READ Menu$(*)
50    DATA Accounts Receivable,Accounts
Payable,Personnel,Payroll,Manufacturing
60    OUTPUT KBD USING "#,B,K";255,"K"            ! Clear the alpha     screen
70    Selection=FNMenu(Menu$(*))
80    PRINT USING "#,K";"You selected option #",Selection,"."
90    END
100   ! ******************************************************************
110 Menu:DEF FNMenu(Menu$(*))
120     GINIT
130     PLOTTER IS CRT,"INTERNAL"
140     WINDOW 0,1,6.5,.5
150     GRAPHICS ON
160     LORG 5
```

```
170     GRAPHICS INPUT IS KBD,"TABLET"
180     Field_length=20
190     FOR I=1 TO SIZE(Menu$,1)
200       MOVE 0,I-.25
210       RECTANGLE .5,.5,EDGE
220       MOVE .25,I
230       LABEL Menu$(I)
240     NEXT I
250     TRACK CRT IS ON
260     REPEAT
270       DIGITIZE X,Y
280       Rounded=INT(Y+.5)
290       Ok=(Rounded>=1) AND (Rounded<=SIZE(Menu$,1)) AND (ABS(Y-Rounded)<.25)
300       Ok=Ok AND (X>0) AND (X<.5)
310       IF NOT Ok THEN
320         BEEP
330         DISP "Please place the cursor in an option box before selection."
340         WAIT 2
350         DISP
360       END IF
370     UNTIL Ok
380     GINIT                   ! Get rid of the cursor
390     GCLEAR                  ! Clear the graphics screen
400     RETURN INT(Y+.5)        ! Return the menu option selected
410   FNEND
```

Color Graphics 5

Color Graphics 5

Color can be used for emphasis, for clarity, and just to present visually pleasing images. Color is a very powerful tool and it follows directly that it is very easy to misuse. Be careful in using color and it will serve as a valuable tool for communication. Misuse it and it will garble the communication.

The biggest benefit of the color computer is that it makes experimenting with color so easy. It is easy to test out ideas before you fully implement them. It is also possible to use the color map for simple animation effects and some just plain impressive images.

Non-Color-Mapped Color

There are two types of color displays available with Series 200/300 computers: color-mapped displays, and non-color-mapped displays. This section discusses the latter type. (Subsequent sections discuss color-mapped displays.)

Specifying a Non-Color-Mapped Display

When either of the following statements are executed:

```
PLOTTER IS CRT,"INTERNAL"
  or
PLOTTER IS 28,"98627A"
```

a non-color-mapped display is selected as the plotting device. With the first statement, the non-color-mapped mode is chosen, even though the display may be capable of operating in the color-mapped mode.

Available Colors

With non-color-mapped displays, eight colors are available through the PEN and AREA PEN statements:

- Black and White
- Red, Green, and Blue (the additive color primaries)
- Cyan, Magenta, and Yellow (the complements of the additive color primaries)

Using the HP 98627A Color Interface and Display

The HP 98627A interface allows you to connect an RGB color monitor to your computer. The HP 98627A does not support color map operations; thus, you cannot change the color of an area on the screen without redrawing that area. Nor can you define your own color-addition scheme as you can with a color-mapped device (see subsequent sections of this chapter). In addition to this, there are only eight pure colors[1]; to get others, you must go to dithering.

If you have an HP 98627A interface connected to a 60 Hz, non-interlaced color monitor[2], you could send graphics to it by executing the following statement:

```
PLOTTER IS 28,"98627A"
```

There are many types of color monitors which you can connect to your computer through an HP 98627A color monitor interface. In the PLOTTER IS statement, you must specify accordingly:

Table 5-1. HP 98627A Display Formats

Desired Display Format	Plotter Specifier
Standard Graphics 512 by 390 pixels, 60 Hz, non-interlaced	"98627A" or "98627A;US STD"
512 by 390 pixels, 50 Hz, non-interlaced	"98627A;EURO STD"
High-Resolution Graphics 512 by 512 pixels, 46.5 Hz, non-interlaced	"98627A;HI RES"
TV Compatible Graphics 512 by 474 pixels, 60 Hz, interlaced (30 Hz refresh rate)	"98627A;US TV"
512 by 512 pixels, 50 Hz, interlaced (25 Hz refresh rate)	"98627A;EURO TV"

---

1 Only eight pure colors can be created on an external color monitor. This is because there is no control over the intensity of each color gun. Each color can be either off or on, and there are three colors: red, green, and blue. Two states, three colors: $2^3=8$.

2 Depending on your choice of color monitor, there may be more specification necessary in the string expression part of the PLOTTER IS statement. See the subsequent table for further information.

The HP98627A's display memory is composed of three "color planes" of as many bits as necessary to compose a full picture. Following is a description of how the various pen selectors affect the operation of an external color monitor.

For the an external color monitor connected through the HP98627A, pen selectors are mapped into the range −7 through 7. Thus:

If pen selector>0 then use PEN (pen selector-1) MOD 7+1
If pen selector=0 then use PEN 0 (complement[1])
If pen selector<0 then use PEN -((ABS(pen selector)-1) MOD 7+1)

Non-Color-Mapped Dominant Pens

The meanings of the different pen values are shown in the table below. The pen value can cause either a 1 (draw), a 0 (erase), n/c (no change), or complement (invert) the value in each memory plane.

Table 5-2. Non-Color-Map Dominant-Pen Mode

Pen	Action	Plane 1 (red)	Plane 2 (green)	Plane 3 (blue)
-7	Erase Magenta	0	n/c	0
-6	Erase Blue	n/c	n/c	0
-5	Erase Cyan	n/c	0	0
-4	Erase Green	n/c	0	n/c
-3	Erase Yellow	0	0	n/c
-2	Erase Red	0	n/c	n/c
-1	Erase White	0	0	0
0	Complement	invert	invert	invert
1	Draw White	1	1	1
2	Draw Red	1	0	0
3	Draw Yellow	1	1	0
4	Draw Green	0	1	0
5	Draw Cyan	0	1	1
6	Draw Blue	0	0	1
7	Draw Magenta	1	0	1

[1] "Complement" means to change the state of pixels; that is, to draw lines where there are none, and to erase where lines already exist.

Choosing Pen Colors

The colors can be selected the same way they are for other display devices—with the PEN statement. If all you are after is highlighting a portion of a graph or chart, this may be all the color that you need. (In non-color-map mode, other graphics displays behave exactly like the HP 98627A Color Interface Card.) The colors and their pen selectors are listed below:

Table 5-3. Default Non-Color-Map Values

Pen Value	Color	Frame Buffer Entry
0	Black	0
1	White	7
2	Red	1
3	Yellow	3
4	Green	2
5	Cyan	6
6	Blue	4
7	Magenta	5

If you are in this mode, you can draw lines in the eight colors listed above. It is possible, however, to fill areas with other shades. These tones are achieved through *dithering.*

Dithering produces different shades by combining dots of the 8 colors described above. The screen is divided up into 4-by-4 cells and patterns of dots within the cells are turned on to match, as closely as possible, the color you specify. Dithered colors are defined with the AREA COLOR and AREA INTENSITY statements. The color models available are discussed in a subsequent section entitled "Color Specification." (The actual color matching process used in dithering is described under "Dithering and the Color Map.") Filling is specified by using the secondary keyword FILL in any of the following statements:

IPLOT	PLOT	POLYGON
RECTANGLE	RPLOT	SYMBOL

GSTORE Array Sizes for the HP 98627A

As mentioned above, different color monitors display different numbers of pixels. To figure the array size necessary to GSTORE an image, multiply the number pixels in the X direction by the number of pixels in the Y direction, multiply that by the number of color planes (three) and divide by sixteen (the number of bits per word). For example, say you want to calculate the array size needed for storing an image created on a U.S. standard monitor (see the first entry in the table above): $512 \times 390 \times 3 \div 16 = 37\,440$ words. However, you cannot specify an array which has any more than 32 767 elements in any dimension. To get around this restriction, what is typically done is to make one dimension the number of memory planes (three) and the other dimension the number of pixels ($512 \times 390 \div 16$). Thus, the statement declaring an array for storing an image from a "U.S. Standard" external color monitor could look like this:

```
INTEGER Image(1:12480,1:3)
```

If your array is larger than necessary to store an image, it will be filled only to the point where the image is exhausted. If your image is larger than your array, the array will be filled completely, and the remainder of the image will be ignored.

The GSTORE and GLOAD statements store the graphics image into this array and load it back into graphics memory, respectively.

Using Color-Map Displays

If you are trying to define a complex human interface, you will need more colors and more control over the colors. The system described in the rest of this chapter (except for dithering) is available only after you turn on the color map. To do so, execute:

```
PLOTTER IS CRT,"INTERNAL";COLOR MAP
```

If you have a 98782A color monitor connected through a 98700 display controller, and it is set to address 25, you could execute:

```
PLOTTER IS 25,"INTERNAL";COLOR MAP
```

In this way, plots can easily be plotted on various devices with a minimum of programming effort.

The Frame Buffer

Most Series 200/300 color displays have bit-mapped color graphics. An area in memory called a *frame buffer* provides 1, 4, 6, or 8 bits of memory for each pixel location. (The number of bits available for describing each pixel is sometimes called the *depth* of the frame buffer.) A 4 bit frame buffer allows each pixel location to contain a number between 0 and 15 (inclusive). Thus a four-plane frame buffer can display lines in 16 different colors on the CRT, simultaneously. At any given time, the values written to the frame buffer fall into four categories:

- *Background Value*—Whenever GCLEAR is executed, all the pixel locations in the frame buffer are set to 0. Thus, 0 is the background color.
- *Line Value*—The PEN statement is used to determine the value written to the frame buffer for all lines drawn. This includes all lines (including characters created by LABEL) and outlines (specified by the secondary keyword EDGE).
- *Fill Value*—The AREA PEN statement is used to specify the value written to the frame buffer for filling areas (specified by the secondary keyword FILL).
- *Dithered Colors*—AREA INTENSITY and AREA COLOR can be used for specifying a fill color, but the results can be surprising when the COLOR MAP option has been selected (see "Dithering and Color Maps"). In addition, the dithered colors have a tendency to introduce texturing into the areas and may not accurately reproduce the color you specify.

The PEN, AREA PEN, AREA INTENSITY, and AREA COLOR statements control what are referred to as *modal attributes*. This means that the value established by one of the statements stays in effect until it is altered by another statement. (GINIT alters all of them.)

Erasing Colors

Erasing is a fairly simple concept in frame buffers that are a single bit deep. You just restore the background by setting the portion of the frame buffer you wish to erase to 0. The concept is a little more complex in frame buffers with more depth. As long as the graphics system is in the dominant writing mode (see "Non-Dominant Writing"), there are three ways of erasing:

- The easiest is GCLEAR. However, GCLEAR destroys the entire image. If you want to erase only part of the image, it is necessary to be more precise.
- If you know the pen used to write the line, you can use a negative pen selector of the same magnitude. This will erase the pen value from the frame buffer. (It works for PEN and AREA PEN.)
- If you don't know the pen used to create the image, you can overwrite the image with the background color. This can be PEN 0, or, if you are on a filled area, whatever pen the area was filled with. A fairly simple extension of this is to use the RECTANGLE statement to implement a *local GCLEAR* to erase portions of the screen.

Default Colors

Throughout the discussion of the frame buffer, only values were talked about. If you do not modify the color map (see the next section for how to do that) the colors selected by the PEN and AREA PEN values depend on the default color map values, which are:

Table 5-4. Default Color Map Values

Value	Color
0	Black
1	White
2	Red
3	Yellow
4	Green
5	Cyan
6	Blue
7	Magenta
8	Black
9	Olive Green
10	Aqua
11	Royal Blue
12	Maroon
13	Brick Red
14	Orange
15	Brown

Pens 16 through the end of the color map are also defined for displays with more than four planes. You can interrogate the color map with GESCAPE for exact values.

The Primary Colors

The lower eight pens of the default color map are the same as are available without enabling the color map, but they do not write the same value into the frame buffer (see the PEN statement in the *BASIC Language Reference*.) The colors are:

- Black and White (the extremes of no-color)
- Red, Green, and Blue (the additive primaries)
- Cyan, Magenta, and Yellow (the complements of the additive primaries - which happen to be the subtractive primaries)

The Business Colors

The upper 8 colors (8 through 15) were selected by a graphic designer to produce graphs and charts for business applications. The colors are:

- Maroon, Brick Red, Orange, and Brown (warm colors)
- Black, Olive Green, Aqua, Royal Blue (cool colors)

These colors are one designer's idea of appropriate colors for business charts and graphs. They were chosen to avoid clashing with each other. A technique for using them is described under "Color Hard Copy" in the "Color Spaces" section at the end of this chapter.

It is possible to use the color computer with the default color map. The color used will depend directly on the value in the frame buffer. This is fine if the work you are doing can be accomplished using the 16 colors supplied as the system defaults. This is often not the case, and this overlooks one of the most powerful features of the color computer—the color map.

The Color Map

The color-mapped system uses the value in the frame buffer as an index into a color map. The color map contains a much larger description of the color to be used and, just as importantly, the color description used is *indirect.* Thus, the value in the frame buffer does not say "use color 12," but rather "use the color described by register number 12." Note that 8-bit binary numbers are stored in the color map in the diagram below. This is not the case for a Model 236C; it stores 4-bit binary numbers. All other color-mapped systems use 8-bit binary numbers in their color map.

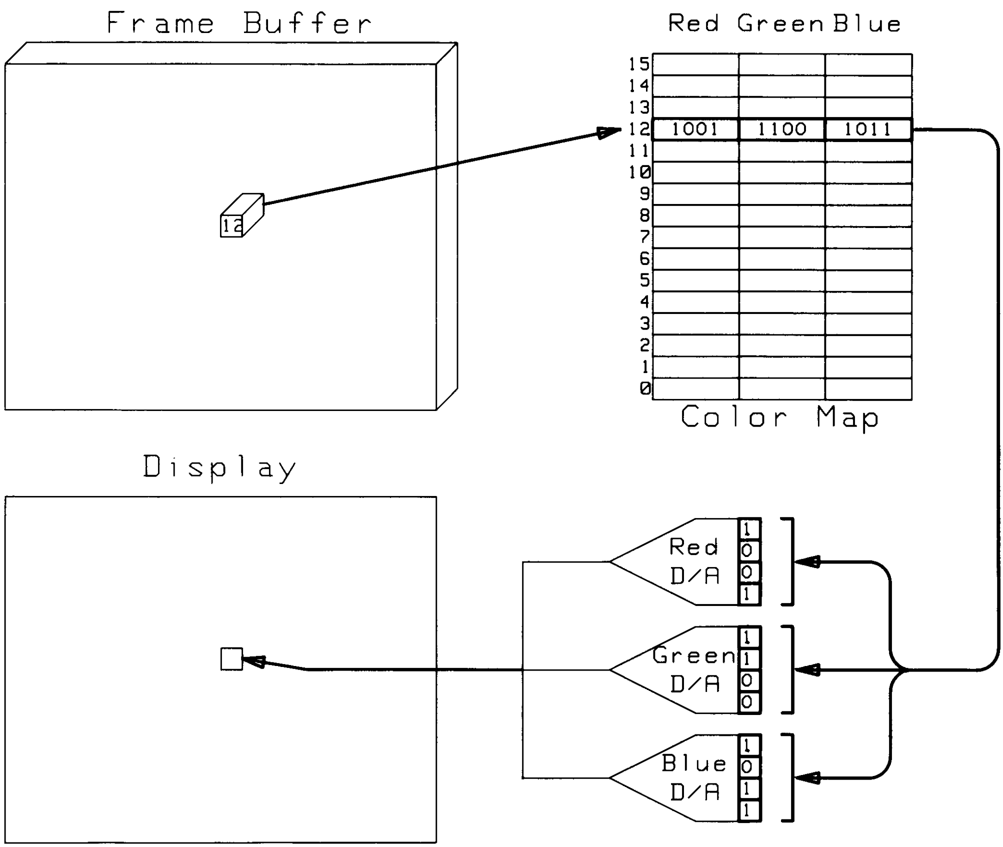


Figure 5-1. Color Map (Model 236C)

The CRT refresh circuitry reads the value from the pixel location in the frame buffer, uses it to look up the color value in the color map, and displays that color at that pixel location on the CRT. Thus, it is possible to draw a picture with a given set of colors in the color map (a set of colors is called a *palette*) and then change palettes and produce a new picture by redefining the colors, rather than having to redraw the picture. (The binary numbers in the color map are created by the system. The user deals with normalized values, as described under "Color Specification.")

Color Specification

The SET PEN statement is used to control the values in the color registers in the color map. The SET PEN statement supports two color models for selecting the color each pen represents, the RGB (Red, Green, Blue) model and the HSL (Hue, Saturation, Luminosity) model. Since the color models are dynamically interactive, it is much easier to understand them by experimenting with them.

The RGB Model (Red, Green, Blue)

The RGB model can be thought of as mixing the output of three light sources (one each of Red, Green, and Blue). The parameters in the model specify the intensity of each of the light sources. The RGB model is accessed through the secondary keyword INTENSITY with the SET PEN statement. The RGB model is closest to the actual physical system used by color displays. The Red, Green, and Blue values represent the value modulating the electron guns that excite the colored phosphors on the CRT. The values are *normalized* (range from 0 through 1). The normalized values are converted to 8-bit[1] binary numbers to store in the color map. Each of the values is used to control a 8-bit[1] digital-to-analog converter, providing 256[1] intensity levels from full-off to full-on for each of the colors. Thus,

```
SET PEN 0 INTENSITY 7/255,7/255,7/255
```

sets pen 0 (the background color) to approximately a "50%" gray value. (Whenever all the guns are set to the same intensity, a gray value is obtained.) It is simpler to think in 1/255ths[2] and let the computer do the conversion to a decimal fraction, since the intensity parameters can be numeric expressions. The parameters for the INTENSITY mode of SET PEN are in the same order they appear in the name of the model (Red, Green, Blue).

[1] The Model 236C stores 4-bit binary numbers in the color map.
[2] The Model 236C uses 1/15ths for its intensity value.

The HSL Model (Hue, Saturation, Luminosity)

The HSL model is closer to the intuitive model of color used by artists, and is very effective for interactive color selection. The three parameters represent *hue* (the pure color to be worked with), *saturation* (the ratio of the pure color mixed with white), and *luminosity* (the brightness-per-unit area.) The following plate is of a screen from the program "NEW_MODELS", and provides a physical model to relate the parameters of the HSL model to.

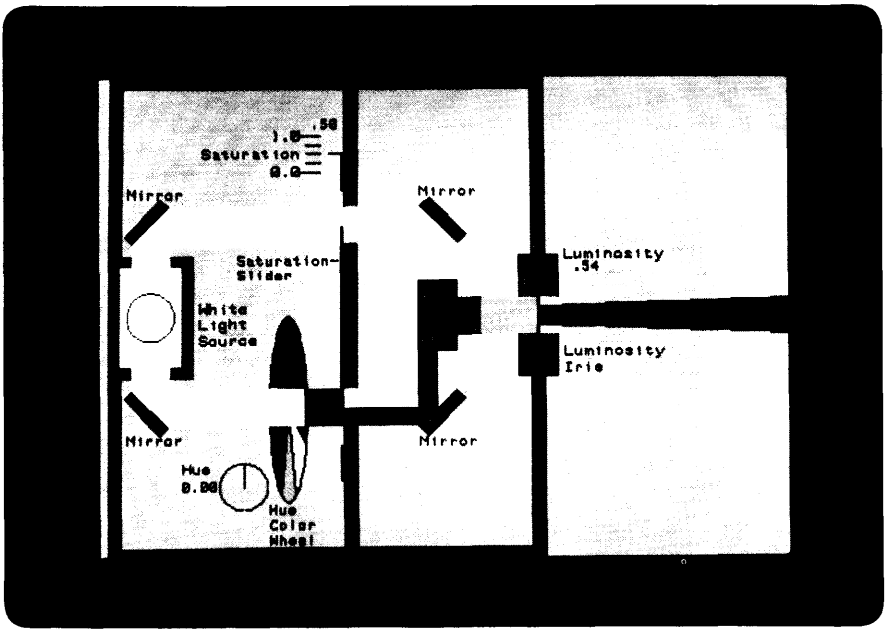


Figure 5-2. HSL Physical Model

The Hue parameter rotates a color wheel to select a "pure" color to use. This color is then mixed with white light. The ratio of the pure colored light to the white light is controlled by the Saturation slider. Finally, the output passes through the luminosity iris (think of it as a hole you can adjust the size of) that controls the brightness of the output.

The HSL model is accessed through the SET PEN statement with the secondary keyword COLOR:

```
SET PEN Current_pen COLOR H(Current_pen),S(Current_pen),L(Current_pen)
```

HSL Resolution

The resolution of the HSL model is not specified anywhere. This is because the resolution for the various parameters is not a fixed value. The resolution for any parameter of the HSL system is dependent on all three of the parameters. The resolution is not only changed by the other two parameters, but also by the magnitude of the parameter you are varying. If resolution of the system becomes important in a program, it is possible to use a GESCAPE to read the RGB values back from the color map to watch for a change in the actual value being written in the color map. Change the HSL parameters by very small increments (on the order of 0.001) until a change in the color map entry is detected. This is best done using color map entry 0, since you will only need to read a single entry from the color map to check for the change.

Which Model?

Two models are provided for your color computer. The INTENSITY option of the SET PEN statement is faster than the COLOR option, because it directly reflects the hardware in the system. If you are working with primaries only, or want gray scale output, the RGB model is great. If, on the other hand, you are trying to deal with pastels and shades, you are better off with a color model that is intuitive in nature, and that is where the HSL model shines.

It is possible to get the best of both worlds by using the HSL model for the human interaction, then reading the color map with a GESCAPE statement to get the RGB color values. The RGB values can then be used to rapidly load a palette into the color map. The "SET_COLOR" program does exactly that to calculate the correct cursor and text color to use when the user sets a background color. This is done by reading in the RGB color map values, calculating which corner of the color cube is farthest from the background color, setting the foreground pen and text displays to that color, and then writing the RGB array back into the color map. Even though the primary interaction is with the HSL model, the RGB is used because it is more convenient to find distances between colors in it.

```
1270  GESCAPE CRT,2;Rgb(*)              ! Read the color map
1280  IF Rgb(0,1)<.5 THEN
1290    Rgb(1,1)=1
1300  ELSE
1310    Rgb(1,1)=0
1320  END IF
1330    !
1340  IF Rgb(0,2)<.5 THEN
1350    Rgb(1,2)=1
1360  ELSE
1370    Rgb(1,2)=0
1380  END IF
1390    !
```

```
1400  IF Rgb(0,3)<.5 THEN
1410    Rgb(1,3)=1
1420  ELSE
1430    Rgb(1,3)=0
1440  END IF
1450    !
1460  Print_color=0
1470  FOR I=1 TO 3
1480    Print_color=Print_color*2+Rgb(1,I)
1490  NEXT I
1500    !
1510  CONTROL 1,5;Funny_number(Print_color)
1520  SET PEN 0 INTENSITY Rgb(*)          ! Refill the color map
```

By the way, lines 1280 through 1490 can be replaced by the following:

```
1280  Print_color=4*(Rgb(0,1)<.5)+2*(Rgb(0,2)<.5)+(Rgb(0,3)<.5)
1290  FOR I=1 TO 3
1300     RGB(1,I)=BIT(Print_color,3-I)
1310  NEXT I
```

These lines will execute faster, but are harder to understand.

One point brought out by the preceding example is that the models can be mixed freely. There is nothing to prevent using INTENSITY to set a gray background color and a black pen, and then using COLOR to produce the rest of the palette. Use whatever is easiest for what you want to do.

If you are interested in pursuing the color models, the RGB model is formally referred to as a color cube and the HSL model is called the Color Cylinder. These models represent idealized color spaces and are described under "Color Spaces" at the end of this chapter.

Dithering and Color Maps

In early color systems which did not provide control of the intensity of individual pixels, dithering became a very popular method of increasing the number of shades available to the machine. By reducing the effective resolution of the system, it was possible to provide a large variety of shades.

Your color computer provides dithering for applications that require more shades than 16, 64 or 256 colors that are available at any single time with the color map on your particular color-mapped system. The AREA INTENSITY and AREA COLOR statements provide access to the dithered colors, although they will fill with a single pen if the color requested exists in the current color map.

If you are not in the color-map mode, AREA INTENSITY and AREA COLOR will produce the same results on a color computer that they produce on non-color-map devices, such as the 98627 Color Interface Card.

Creating A Dithered Color

The following discussion gets a little abstract, and it is not absolutely necessary to understand how dithering works to use it. It is interesting information and can be useful knowledge if dithered areas don't do what you expect.

A color vector is a directed line connecting two points in RGB color space. The dithering process tries to match a *target vector* by constructing a *solution vector* from colors in the color map. The actual dithered color to be produced will be 16 times the target vector, since 16 points in the dither area will be combined to create it.

The color matching process requires sixteen steps. First, the target vector is compared to the vectors produced by all the colors in the color map. The one which is closest[1] to the target vector is selected as the first component of the solution vector.

---

[1] The distance between the points in the RGB color space is used. The RGB color space is a 3-dimensional Cartesian coordinate system.

The following process is then repeated 15 times:

1. The target vector is added to itself to produce a new target vector.
2. A trial solution vector is created for each color in the color map by adding the vector for the color map entry to the previous solution vector. The trial solution vector that is closest to the target vector is selected as the new solution vector.

At this point, the target vector is 16 times the original target vector, and the solution vector consists of a summation of color vectors from the color map that produce, at each iteration, the vector closest to the target vector.

The colors are then sorted by luminosity and filled into the following precedence matrix (the most luminous color is filled into the lowest numbered pixel):

Table 5-5. Dithering Precedence Matrix

1	13	4	16
9	5	12	8
3	15	2	14
11	7	10	6

The dither precedence matrix is actually tied to pixel locations on the CRT. The matrix is repeated across the CRT and from the top to the bottom of the CRT. Areas to be filled are mapped against the fixed dithering pattern. All dither cells completely within an outline to be filled are turned on according to the precedence pattern. Cells which are only partially within the border are only partially enabled. If the area fill pattern calls for a pixel outside the boundary to be set, it will not be.

There are problems with dithering, especially when used with the color map:

- The dithered color selection tends to produce textures. In some cases, the textures overwhelm the shade produced.
- The dithered colors are not necessarily accurate representations of the color specified. This is especially true if the color map is loaded with a palette that is less than ideal for dithering. A 4-by-4 dither cell with one full intensity green pixel does not look the same as the same cell filled with the color map color 1/15 green.

- The dithered colors are not stable if the color map is altered. (If you change the color map after doing a fill based on an AREA COLOR or AREA INTENSITY, the fill value can change.)
- The dithering operation produces anomalies when the area to be filled is thin. If it is less than four pixels wide or high, it cannot contain the entire dither cell and the results can be surprising for colors which turn on small portions of the cell.

If You Need More Colors

If you have an application that requires more than 16 (4-plane systems), 64 (6-plane systems) or 256 (8-plane systems) colors, the first thing to do is see if you can redefine it to use 16, 64 or 256 colors. In many cases this is possible, and the higher quality of the color mapped palette is worth a little checking to see if you can use it.

If you absolutely have to get at a larger palette, then load a palette optimized for dithering (optimizing for dithering is described below) and **stick with dithering**. Don't try to mix color map redefinition and dithering—it will probably cause you a lot of grief. Especially, do not try to do interactive redefinition of the color map in a system that also does dithering.

Optimizing for Dithering

The actual color palette you require determines the optimum color map values. The following program leaves the additive primaries and their complements in the lower eight locations and replaces the designer colors in the upper half of the color map with half-luminosity values for each of the lower eight colors.

```
10     ! "DITHER_PAL"
20     ! This program creates a palette for dithering
30     ! on a 16 color map system.
40     GINIT
50     PLOTTER IS CRT,"INTERNAL";COLOR MAP
60     GRAPHICS ON
70     WINDOW 0,16,-.1,1
80     DIM Colors(0:15,1:3)
90     GESCAPE CRT,2;Colors(*)
100    FOR I=0 TO 7
110      Colors(I+8,1)=Colors(I,1)/2
120      Colors(I+8,2)=Colors(I,2)/2
130      Colors(I+8,3)=Colors(I,3)/2
140    NEXT I
150    SET PEN 0 INTENSITY Colors(*)
160    FOR I=0 TO 15
170      MOVE I,0
180      AREA PEN I
190      RECTANGLE 1,1,FILL
```

```
200   NEXT I
210   END
```

The color map generated by "DITHER_PAL" is optimized for producing the widest selection of colors. If you have specialized needs you can create palettes that are even more optimum for specific applications. For example, you could load the color map with 16 shades of red to produce an optimum palette for producing an image that only contained red objects.

Non-Dominant Writing

All the techniques described up until now have dealt with dominant writing to the frame buffer. In the dominant writing mode, the pen selector is written directly to the color map, and overwrites whatever is currently in the frame buffer. In non-dominant writing, a bit-by-bit logical-OR is performed on the contents of the frame buffer and the pen selector value being written to the frame buffer. Thus, if pen 1 is written to a buffer location that has already been written to with pen 6, the buffer location will contain 7, but writing pen 2 to a buffer location that has already been written to with pen 6 will not change the contents.

Non-dominant writing can be used to create a properly defined palette of colors in the color map. Using this palette of colors, it is fairly easy to create a copy of the primary color circles. An additive palette is created in lines 490 through 540, by modeling the three least-significant bits of the frame buffer as *color planes*. Bit 0 is treated as representing red, bit 1 as representing green, and bit 2 as representing blue.

```
470   !****************** Create the Additive Palette ***
480   !
490   FOR I=0 TO 7
500     Red=BIT(I,0)
510     Green=BIT(I,1)
520     Blue=BIT(I,2)
530     SET PEN I INTENSITY Red,Green,Blue
540   NEXT I
```

The palette is created in the color map and then read into an array, using GESCAPE.

```
620   GESCAPE CRT,2;Additive(*)    ! Read additive palette
```

The subtractive palette is created in lines 750 through 840. The palette is created by converting between the RGB map created for the additive palette, above, and a CMY (Cyan, Magenta, and Yellow) system. (The technique is described in more detail in the next section, "Color Spaces.")

```
750   FOR I=0 TO 15                      ! Create subtractive palette
760     FOR J=1 TO 3
770       Point(1,J)=Additive(I,J)       ! Read a point from additive palette
780     NEXT J
790     MAT New_point= Unit-Point
800     !
810     ! The next line prints out PEN INTENSITY values for both palettes
820     IF I<8 THEN PRINT USING Pen_image2;White$,I,Point(1,1),Point(1,2),
Point(1,3),Black$,I,New_point(1,1),New_point(1,2),New_point(1,3)
830     SET PEN I INTENSITY New_point(*) !
840   NEXT I
```

The Surprise palette is created by reading from data statements.

```
210   !************** Create the Surprise Palette ****************
220   !
230   SET PEN 0 INTENSITY .6,.6,.6          ! Gray background
240   RESTORE Surprise                      ! Make sure you read the right data
250 Surprise:  ! DATA for surprise palette
260   DATA .9          ! Pen 1
270   DATA .2          ! Pen 2
280   DATA .5          ! Pen 3
290   DATA .7          ! Pen 4
300   DATA .1          ! Pen 5
310   DATA .8          ! Pen 6
320   DATA .3          ! Pen 7
330   !
340   FOR I=1 TO 7
350     READ Hue
360     SET PEN I COLOR Hue,1,1
370   NEXT I
380   !
390   MAT Point= (.6)                       !\
400   SET PEN 8 INTENSITY Point(*)          ! \
410   SET PEN 9 INTENSITY Point(*)          !  > Pens for labels
420   MAT Point= (0)                        ! /
430   SET PEN 10 INTENSITY Point(*)         !/
440   !
450   GESCAPE 3,2,Surprise(*)
```

The surprise palette relates to no known color system, but it demonstrates an important point—*the non-dominant color map is arbitrary, and can represent any system you can dream up.* You may want to write in four shades of blue, have any overlap of two pens be yellow, any overlap of three pens be orange, and any overlap of four pens be red. The following lines set up such a color map.

```
230   DIM Yellow(1:1,1:3),Orange(1:1,1:3)
240   RESTORE Colors
250   READ Yellow(*),Orange(*)
260 Colors:DATA .87,.87,0,        1,.47,0
270          !
280   SET PEN 0 INTENSITY .6,.6,.6      ! Gray background
290   SET PEN 1 INTENSITY 0,0,.4        ! 0001 - Blue Plane 1
300   SET PEN 2 INTENSITY 0,0,.6        ! 0010 - Blue Plane 2
310   SET PEN 3 INTENSITY Yellow(*)     ! 0011
320   SET PEN 4 INTENSITY 0,0,.8        ! 0100 - Blue Plane 3
330   SET PEN 5 INTENSITY Yellow(*)     ! 0101
340   SET PEN 6 INTENSITY Yellow(*)     ! 0110
350   SET PEN 7 INTENSITY Orange(*)     ! 0111
360   SET PEN 8 INTENSITY 0,0,1         ! 1000 - Blue Plane 4
370   SET PEN 9 INTENSITY Yellow(*)     ! 1001
380   SET PEN 10 INTENSITY Yellow(*)    ! 1010
390   SET PEN 11 INTENSITY Orange(*)    ! 1011
400   SET PEN 12 INTENSITY Yellow(*)    ! 1100
410   SET PEN 13 INTENSITY Orange(*)    ! 1101
420   SET PEN 14 INTENSITY Orange(*)    ! 1110
430   SET PEN 15 INTENSITY 1,0,0        ! 1111
440   !
450   GESCAPE 3,2,Surprise(*)
```

Backgrounds

One nice feature available with non-dominant writing is backgrounds that aren't altered by your foreground. By restricting your foreground to pens 0 through 7, a background written with pen 8 will not be damaged by writing over it.

Complementary Writing

The concept of complementary writing was introduced in the "Interactive Graphics" chapter, under "Making Your Own Echoes." On a color computer, the concept of a complementary pen is extended to deal with the 4-bit or 8-bit values in the color map. With the non-dominant writing mode enabled, negative pen numbers will be exclusively-ORed with the contents of the frame buffer.

The complement occurs only for the bits which are one in the pen selector. Thus a pen selector of −6 would complement bits 1 and 2 of the frame buffer. If a 1 exists in a frame buffer location and a line is drawn over it with PEN −6, a 7 will now be in the location. Writing over the pixel with the same pen selector will return it to a 1.

Making Sure Echoes Are Visible

It is important to understand that the complementing is of the frame buffer, not the color map. You are responsible for making sure that the complemented frame buffer values are visible against one another. Be careful of placing the same color in two locations on the color map that are complements of one another. If you pick one of them as an echo color and then try to use the echo over an area filled with the other value, you will not be able to see it.

Effective Use of Color

At the beginning of this chapter, it was pointed out that color is a very powerful tool, and that it was also easy to misuse. While it is beyond the scope of this book to provide an exhaustive guide to color use, a few comments can be made on using color effectively.

This section will deal with seeing color first, to lay the groundwork. This is followed by a discussion on designing effective display images, since effective color use is almost impossible if the image is fundamentally unsound.

After laying the groundwork, effective color use is discussed, from both the objective and subjective standpoints.

Seeing Color

The human eye responds to wavelengths of electromagnetic radiation from about 400 nm to about 700 nm (4000 to 7000 angstrom.) We call this visible light. Visible light ranges from violet (400 nm) to red (700 nm.) If all the frequencies of visible light are approximately equally mixed, the result is called white light.

The eye's ability to discriminate color is reduced as the light level is reduced. This means that the variety of colors perceivable at low light levels is smaller than the variety at higher light levels.

The eye is most sensitive to colors in the middle of the visible spectrum, a yellow-green color. The eye is least sensitive to the shorter wavelengths, which are at the blue end of the spectrum. Sensitivity to red is between that of yellow-green and blue. Two things seem to be associated with the sensitivity of the eye to various colors:

- The eye can distinguish the widest range of colors in the yellow-green region, and the smallest variety of colors in the blue region.
- The eye is most sensitive to detail in the yellow-green region.

Why and how any of the above works is explained by color theorists. There are a large number of theories of color and all of them work for explaining the specific phenomena the researchers were studying when they developed the theory, but none of them seem to be able to explain it all. The list of references at the end of this chapter include several on how vision works.

It's All Subjective Anyway

One of the reasons that there are so many color theories is that no two people seem to perceive color the same way. In fact, the same person will many times perceive color differently at different times. In addition to the physiological and psychological variables in color perception, many environmental factors are important. Ambient lighting and surrounding color affect the perceived color tremendously.

Mixing Colors

If two distinct audio tones are played simultaneously, you will hear both of them. If the same area is illuminated by two or more different colors of light, you will not perceive the original colors of light, but rather a single color, and it will be not be one of the original colors. What you will perceive is called the *dominant wavelength.*

The CRT uses three different colored phosphors (Red, Green, and Blue) and mixes various intensities of the resulting lights to produce one of XXXX colors at any point on the CRT. What you actually see is the resulting dominant wavelength. This is an additive color system.

Mixing with pigments is a little different. Pigments in inks and paints absorb light. The idea with pigments is to subtract all but the color you want out of a white light source. This is a subtractive color system, and the primary colors are cyan, magenta, and yellow.

The different mechanisms for mixing additive and subtractive colors make it difficult to reproduce images created with additive colors (like a CRT) in a subtractive medium (like a plotted or printed page.) Photographing the CRT is the best method currently available for color hard copy. This problem is discussed in more depth at the end of this chapter under "Color Gamuts" and "Color Hard Copy."

Designing Displays

While the design of displays is not really a color topic, a few words about it are in order before we get into the effective use of color. If the design of an image is fundamentally unsound, all the good color usage in the world is not going to help it.

Whenever you put an image on a CRT, you have created a graphic design. The design will either be a good one or a bad one, and if you know this, you have automatically increased your chances of creating a good design. If you are going to be creating a lot of displays, either in a lot of programs or in a single large program, you need a graphic designer. Many people have a natural talent for graphics —an ability to look at an image and tell whether it is graphically sound or not. If you don't have that talent (or feel you could use some help) there are two courses of action that might be productive for you; you can hire a graphics designer or become one. Renting one is expensive and becoming one is time-consuming, but if you are trying to communicate with users, *you have to understand graphic design.* While getting a degree in graphic arts may be impractical, a course or two in the field will prove very useful if you do much programming.

While this book can't turn you into a graphic designer, a few simple hints may help you on your next program.

The most important thing in communicating with people is to keep it simple. Don't try to communicate the total sum of human knowledge in a single image. It is much more effective to have several screens of information that a user can call up as required than a single screen so complicated that the user can't find what he wants on it.

Try to redundantly encode everything in case one of the encoding methods fails. For example, if you color code information, use positional coding (the location of the information tells something about the nature of the information) too. Remember, the person reading the screen is probably *not* the person who wrote the program, and even if you are writing the program for yourself, you may forget how it works by the next time you try to use it.

Another important thing to remember is to be consistent. Always try to place the same type of information in the same area of the CRT and use the same encoding methods for similar messages. Don't using flashing to encode important information on one display and then using inverse video for the same thing seven displays into the program.

Objective Color Use

In spite of the subjectivity of color, there are some fairly objective things that you should know about color. Some of the things that can be done with color don't depend heavily on subjective interpretation.

Color Blindness

A fact of life that it is dangerous to ignore is that some people are color-blind. The most common form of color blindness is red-green color blindness (the inability to distinguish red and green). Avoid encoding information using red-green discrimination, or these people will have difficulty using the system.

Color Map Animation

One very powerful communication tool is motion. Some simple forms of animation can be achieved by changing the colors in the color map. This technique of color map animation is capable of adding simple motions to an image. Color map animation can be combined with frame buffer animation, which is based on creating images and storing them, to produce more dramatic animated effects.

The basic technique of color map animation can be broken into 3 steps:

1. Create the palettes (or starting palette.)
2. Create the image.
3. Load or modify the palette to add motion.

A look at a simple program example will help show how color map animation works. Load and run "Marquee" from the *Manual Examples* disc. The moving color bands around the label are not redrawn to produce the motion—the color they represent is changed in the color map. Let's look at each section of the program to see how color map animation works.

The first step is declaring some arrays. Most of the arrays will hold RGB pen values to use with SET PEN to define new color palettes. `Black` contains all black pens, so the image can be drawn without being seen. `Message$` is used to hold strings to print on the alpha screen while the image is being created. `Pal1` through `Pal4` are palette arrays that contain the color maps for the animation. `New_order` will be used to create the palette arrays.

```
10    ! "MARQUEE"  - a demo of color map animation
20    !
30    DIM Black(0:15,1:3),Message$[80]
40    DIM Pal1(1:6,1:3),Pal2(1:6,1:3),Pal3(1:6,1:3),Pal4(1:6,1:3)
50    INTEGER New_order(1:6)
```

The first three lines in the following block of code are used to put a message on the alpha screen for the person running the program. It takes several seconds for the program to set up the animation. Messages are printed on the screen to keep the viewer from getting bored.

The next step is to create the palettes. The palette will be loaded into pens 1 through 6. An initial palette is read into `Pal1` from data statements. Pens 1 through 4 will be used for the actual animation. These are red, green, blue, and black. The black band is necessary to produce a strong illusion of motion. The other colors can be whatever you want.

Pen 5 provides a stable background to label the marquee message in, and pen 6 is used for two purposes:

- Each rectangle is framed with a fixed pen to provide reference points for the motion. Perception of motion is relative, and the illusion is much more pronounced when the rectangles are framed.
- The message in the marquee is labeled in a fixed pen, to make it easy to read.

Once the initial palette is loaded, MAT REORDER is used to rearrange the colors, rotating them by one position in each successive palette. Only the lower four pens are rotated.

```
60    OUTPUT KBD USING "#,B";255,75          ! Clear alpha screen
70    ALPHA ON                               ! Obvious
80    PRINT "What's that?"                   ! Give them something to read
90    Pause_time=.084                        ! Display each palette this long
100   MAT Black= (0)                         ! All pens black
110   RESTORE Pal                            ! Read the right data
120   READ Pal1(*)                           ! Read the base palette
130   READ New_order(*)                      ! Read the reordering vector
140 Pal:DATA 1,0,0,   0,1,0,   0,0,1,     0,0,0,  0,0,0,  0,1,1
150   DATA 2,3,4,1,5,6
160   MAT Pal2= Pal1                         !\
170   MAT REORDER Pal2 BY New_order          ! \  Copy preceding palette
180   MAT Pal3= Pal2                         !  \ and reorder the lower
190   MAT REORDER Pal3 BY New_order          !  / four entries to rotate
200   MAT Pal4= Pal3                         ! /  the colors for the
210   MAT REORDER Pal4 BY New_order          !/   lower four pens.
```

Next, we set up the graphics system. It *must* be in the color map mode. The scaling was set up to be convenient for generating the border of bars. The scaling allows for softkeys to be included under the image.

```
220   GINIT                                  ! Set defaults
230   PLOTTER IS CRT,"INTERNAL";COLOR MAP    ! Set color map
240   SET PEN 0 INTENSITY Black(*)           ! All pens black
250   GRAPHICS ON                            ! Obvious
260   WINDOW 0,30,-3,30                      ! Arbitrary scale
270   PEN 6                                  ! Border and text pen
```

A set of concentric rectangles are generated with the RECTANGLE statement, framed (EDGE) with pen 6 (one of the stable colors) and filled (FILL) with one of the pens (1 through 4) that will be used for the animation. The inner rectangle is filled with pen 5 to provide a stable background for the labels. Messages are read from data statements and printed on the screen to keep the viewer's attention.

```
280   RESTORE Text                           ! Read the right data
290   FOR I=1 TO 9                           ! 8 nested rectangles
300     AREA PEN I MOD 4+1                   ! Use pens 1 through 4
310     IF I=9 THEN AREA PEN 5               ! Inner rectangle for message
320     MOVE (0+I*.5),0+I*.5                 ! Corner of the rectangle
330     RECTANGLE (30-I),30-I,FILL,EDGE      ! Draw a filled rectangle
340     IF I MOD 2 THEN                      !\   Print a message after
350       READ Message$                      ! \  every other rectangle;
360       PRINT Message$                     ! /  (Don't let them get
370     END IF                               !/    bored while setting up.)
380   NEXT I
390 Text:DATA "You're tired of the same old computer programs?"
400   DATA "Ready for something new?","Don't Move.","Don't Go Away."," "
```

Now we add the text in the marquee. The delay in line 530 is for dramatic effect.

```
410   CSIZE 10                               !\ Set up for the labels
420   LORG 5                                 !/
430   MOVE 15,17                             ! Location for labels
440   LABEL USING "K";"Coming soon"          !\
450   LABEL USING "K";"To a Model 36C"       ! > Labels in Marquee
460   LABEL USING "K";"Near You."            !/
470   FOR I=-.04 TO .04 STEP .01             !\
480     MOVE 15+I,22                         ! \  Make this label bold
490     LABEL USING "K";"The Tiger"          ! /
500   NEXT I                                 !/
510   OUTPUT KBD USING "#,B";255,75          ! Clear the Alpha screen
520   PRINT "It's time for:"                 ! Last text message
530   WAIT 2                                 ! Let them read it
540   OUTPUT KBD USING "#,B";255,75          ! Clear the Alpha screen
```

The following code begins the actual animation. The palettes are loaded in succession to create the motion effect. Varying the value of `Pause_time` (defined in line 90) changes the speed of the apparent motion. Since each palette is a single positional rotation of the preceding palette, and the last palette looks like it is one rotation away from the first palette, we can simply loop back to the first palette.

```
550   LOOP                                      ! Do forever
560     SET PEN 1 INTENSITY Pal1(*)             !\
570     WAIT Pause_time                         ! \
580     SET PEN 1 INTENSITY Pal2(*)             !  \
590     WAIT Pause_time                         !   \ Load the four palettes,
600     SET PEN 1 INTENSITY Pal3(*)             !   / waiting after each load.
610     WAIT Pause_time                         !  /
620     SET PEN 1 INTENSITY Pal4(*)             ! /
630     WAIT Pause_time                         !/
640   END LOOP
650   END
```

Study this program segment to conceptualize a technique for color animation.

A color wheel is animated using a similar technique, except that the color map is calculated each time, rather than being a pre-calculated set of values.

```
9080 Make_color_pens:!
9090  Wheel_hue(11)=Hue-4*Del_hue
9100  IF Wheel_hue(11)<0 THEN Wheel_hue(11)=1+Wheel_hue(11)
9110  Wheel_hue(10)=Hue-3*Del_hue
9120  IF Wheel_hue(10)<0 THEN Wheel_hue(10)=1+Wheel_hue(10)
9130  Wheel_hue(9)=Hue-2*Del_hue
9140  IF Wheel_hue(9)<0 THEN Wheel_hue(9)=1+Wheel_hue(9)
9150  Wheel_hue(8)=Hue-Del_hue
9160  IF Wheel_hue(8)<0 THEN Wheel_hue(8)=1+Wheel_hue(8)
9170  !
9180  Wheel_hue(7)=Hue
9190  Wheel_hue(6)=(Hue+Del_hue) MOD 1
9200  Wheel_hue(5)=(Hue+2*Del_hue) MOD 1
9210  Wheel_hue(4)=(Hue+3*Del_hue) MOD 1
9220  !
```

In addition, the palette is loaded in ascending order (pen 1 first, then pen 2, etc.) to rotate the wheel in one direction and in descending order to rotate in the other direction.

```
9230  IF Hue_up=True THEN
9240    FOR Ij=4 TO 11
9250      SET PEN Ij COLOR Wheel_hue(Ij),1,1
9260    NEXT Ij
9270  ELSE
9280    FOR Ij=11 TO 4 STEP -1
```

```
9290      SET PEN Ij COLOR Wheel_hue(Ij),1,1
9300    NEXT Ij
9310  END IF
```

The speed at which the wheel can be rotated is limited by the computation and by the fact that the HSL model is used.

"RIPPLES" and "STORM" (on the *Manual Examples* disc) are two more examples of color map animation.

3D Stereo Pairs

The program "STEREO" on the *Manual Examples* disc demonstrates a method for viewing three-dimensional information on a two dimensional display device. The program produces a pair of images on the CRT. The two images form what is called a *stereo pair*. The stereo pair consists of an image representing the scene as it would be seen by the left eye and one representing the scene as it would be seen by the right eye. When the two images are viewed correctly, the brain merges them together into a single, three-dimensional image.

One of the easiest ways to do that is to use different colored images and then put matching filters over the eyes.

In "STEREO", one image is written in red and the other in blue. A red filter should be placed over the left eye and a blue filter over the right eye. This is the same arrangement used for broadcasting stereo movies over NTSC (American) color television.

The filters normally available for viewing television stereo transmissions are not very narrow, and some "ghosting" occurs (faint images intended for one eye visible in the other.) Narrower filters would actually be better. The CIE coordinate ranges for each of the phosphors are listed below:

Table 5-5.

Color	X Range	Y Range
Red	0.620 through 0.640	0.325 through 0.350
Green	0.280 through 0.315	0.600 through 0.673
Blue	0.150 through 0.153	0.055 through 0.062

If you don't want to get into CIE coordinates, borrow a set of filters, and look for two that produce the images with the least ghosting from the other color. Those are the two you want to use.

The images are written in a non-dominant mode, with a palette set up to allow the intersection of the two images to be visible in both eyes.

The program could be improved by using true perspective, instead of view-plane projection to produce the images.

Subjective Color Use

Choosing appropriate colors for a program to use can be tricky, and constitutes a significant part of the job of a good graphic designer. In the final analysis, it is a largely a matter of trying combinations until you come up with a set of colors that look good together. If your application is complex, it will be well worth your while to consult with a graphic designer about the color scheme and layout of information displays for your program. There are, however, a few fairly fundamental things to remember in designing your programs.

Choosing Colors

First, and probably most important, is to use color sparingly. Color always has a communication value and using it when it carries no specific information adds noise to the communication.

Use some method for selecting the colors–one of the best is a color wheel (see the SET PEN entry in the *BASIC Language Reference*).

- Try varying the luminosity or saturation of a color and its complement (opposite it on the color wheel).
- Try color triplets (three equally-spaced colors) and other small sets of colors equally-spaced around the color wheel.

Pastels (less than fully-saturated colors) tend not to clash.

Give careful attention to your background color. Remember that a filled area can become the background color for a portion of the image on the CRT.

- If you are using a small number of colors, use the complement of one of them for the background.

- If you are using a large number of colors, use a gray background.

If two colors are not harmonious, a thin black border between them can help.

Use subtle changes (such as varying the saturation or luminosity of a hue) for differentiating subtly different messages and major changes (such as large changes in the hue of saturated colors) to convey major differences.

Most of all, think and experiment. The final criteria is "Does this display communicate the message?"

Psychological Color Temperature

Temperatures ranging from cool to hot are associated with colors ranging from blue to red (ice blue–fire red). This is actually the opposite of physical reality, where the higher the temperature, the shorter the wavelength (blue is a black body radiation of about 7600° K while red is about 3200° K) but this is what people *perceive* as the relation between temperature and color. This is probably because people very seldom deal with the high temperatures and associate the blues with non-temperature related natural phenomena (oceans and ice). If you are trying to portray temperature, electrical field strength, stress, or some other continuous physical system, using the psychological color temperature can serve as a useful starting point for color coding the values.

Cultural Conventions

When trying to use color for communicating, cultural conventions are useful. Red is widely associated with danger in most western cultures, giving extra emphasis to a flashing red indicator. By the same token, a flashing green indicator would be less effective for communicating an out of range value in a system. In any specific application, it is important to understand the color associations that are common for the group using the application.

Color Spaces

If you ask a broadcast engineer what the primary colors are, he will probably tell you "Red, green, and blue." If you ask a printer what the primary colors are, he will probably tell you "Cyan, magenta, and yellow." If you ask a physicist, he will probably smile and say "That's not the right question." Let's see if we can get enough information about color systems to ask the right question.

Primaries and Color Cubes

The reason is that there are two sets of color primaries. Red, green and blue are additive primaries. Cyan, magenta, and yellow are subtractive primaries. Each of these sets of primaries can be used to construct what is referred to as a *color cube.* These are called the RGB color cube and the CMY color cube.

Each of the color cubes can be used to describe a *color space.* Color spaces are mathematical abstractions which are convenient for scientific descriptions of color. This is because the color spaces provide a coordinate system for describing colors. Once you have a coordinate system, you can talk about and manipulate colors mathematically.

In addition to the color cubes, other color coordinate systems exist. While there are many, we will only look at HSL Color Space, because it is one of the available color models on your color computer. First, the cubes.

The RGB Color Cube

The RGB color cube describes an *additive* color system. In an additive color system, color is generated by mixing various colored light sources. (Color mixing is discussed in "Effective Use of Color," above.)

The origin (0,0,0) of the RGB color cube is black. Increasing alues of each of the additive primaries (Red, Green, and Blue) move towards white (the opposite corner of the cube.) The maximum for all three colors is white (1,1,1).

A diagonal of the cube connecting (0,0,0) and (1,1,1) represents gray shades, which are generated by incrementing all three color axes equally.

The RGB color cube can be accessed directly, in 16 steps (4-plane systems) or 256 steps (8-plane systems) for each axis, by the INTENSITY option for the color definition statements (SET PEN, AREA INTENSITY, and AREA COLOR).

The CMY Color Cube

The CMY color cube represents a subtractive color system. In a subtractive color system, colors are created by subtracting colors out of a pure white (containing all colors equally) light source. This most often occurs when light is reflected off of surfaces containing or coated with pigments. This happens in printing and painting, among other operations.

The origin (0,0,0) for the CMY color cube is white. This represents all the colors in a perfect white (containing all colors) light source being reflected by a white (reflecting all colors) surface. Increasing values of each of the subtractive primaries (Cyan, Magenta, and Yellow) move towards black (the opposite corner of the cube). The maximum for all three colors is black (1,1,1).

A diagonal of the cube connecting (0,0,0) and (1,1,1) represents gray shades, which are generated by incrementing all three color axes equally.

Converting Between Color Cubes

It is sometimes useful to convert from one color coordinate system to another.

The CMY color cube can be converted to RGB coordinates (or RGB to CMY) by producing a color triplet (a 1 by 3 matrix) containing the CMY coordinate and subtracting this from a color triplet representing a unit color vector (1,1,1). This operation represents rotating the color cube to bring the CMY black (1,1,1) to the RGB black (0,0,0).

The following program lines convert the RGB color map into CMY values. This is done to provide separations of an RGB image into CMY values for printing (remember—printing is a subtractive process). Since the system is color mapped, you only need to convert 16 (4-plane systems), 64 (6-plane systems) or 256 (8-plane systems) values—remember, the frame buffer values only point to a register in the color map.

- The contents of the color map are copied into `Old_colors` using a GESCAPE in line 14680.
- Each color triplet in the color map is copied into `Rgb_point` in lines 14720 through 14740.
- The actual RGB to CMY conversion is done in line 14750.
- The CMY triplet is copied into the CMY array in lines 14760 through 14780.

```
14660 Convert_colors:!
14670   ALPHA ON
14680   GESCAPE 3,2;Old_colors(*)
14690   PRINT "       OLD COLORS                   NEW COLORS"
14700   PRINT "Index  R     G     B            C     M     Y"
14710   FOR I=0 TO 15
14720     FOR J=1 TO 3
14730       Rgb_point(J)=Old_colors(I,J)
14740     NEXT J
14750     MAT Cmy_point= Unit_point-Rgb_point
14760     FOR J=1 TO 3
14770       New_colors(I,J)=Cmy_point(J)
14780     NEXT J
14790     PRINT USING Image$;I,Rgb_point(1),Rgb_point(2),Rgb_point(3),
          Cmy_point(1),Cmy_point(2),Cmy_point(3)
14800   NEXT I
14810   Converted=True
14820   RETURN
```

A subprogram can be used to provide drivers to produce monochromatic gray-scale displays representing the cyan, magenta, and yellow contents of the color map (and a separate black image that printers like to have around). The monochromatic representation is easier to photograph than the actual color content.

This color conversion just described is mathematical. If you really want to print it, you will have to work with a printer to calibrate the frames you are giving him against a good color photo of the actual image. The printer may also want the CMY information to be inverted for his process. This can be achieved photographically or by subtracting each of the CMY values from one during the color map conversion (this is an element-by-element subtraction, not a matrix subtraction). The conversion can be achieved easily with the MAT statement:

```
14805   MAT New_colors = (1) - New_colors
```

HSL Color Space

The color cubes are very useful for working with physical systems that are based on color primaries. They are not always intuitive, though.

The HSL color cylinder resides in a cylindrical coordinate system. A cylindrical coordinate system is one in which a polar coordinate system representing the X-Y plane is combined with a Z-axis from a rectangular coordinate system.

- The coordinates are normalized (range from 0 through 1).
- **Hue** (H) is the angular coordinate.
- **Saturation** (S) is the radial coordinate.
- **Luminosity** (L) is the altitude above the polar coordinate plane.

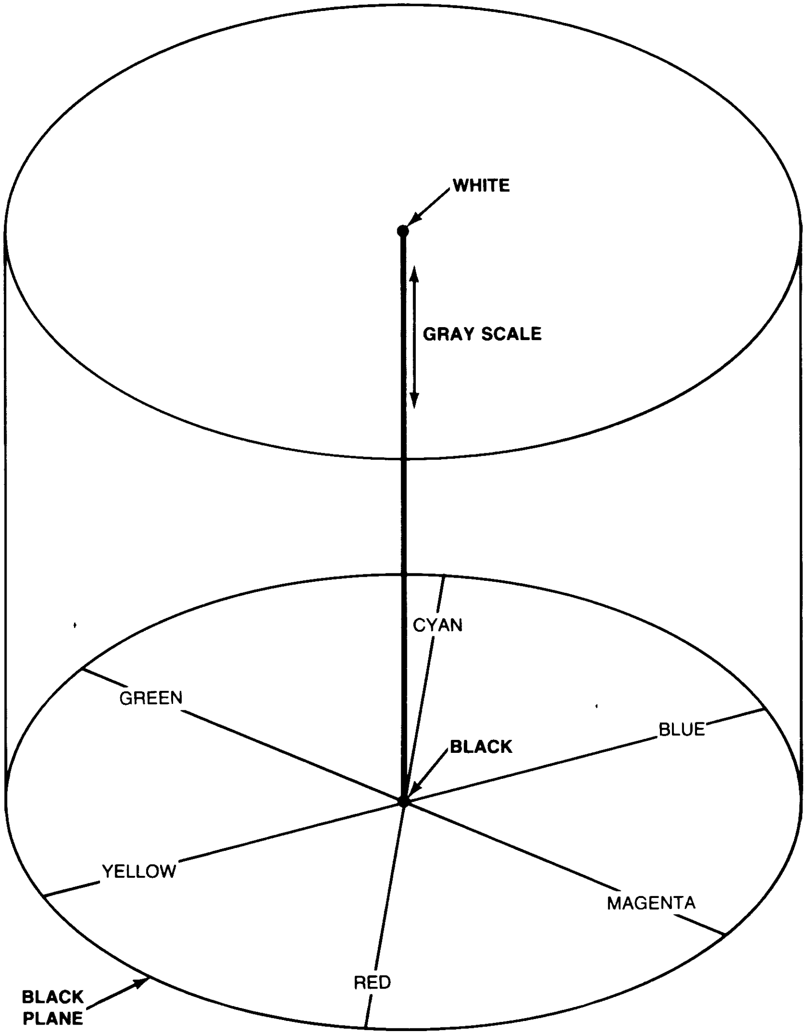


Figure 5-3. HSL Color Cylinder

The cylinder rests on a black plane (L = 0) and extends upward, with increasing altitude (Luminosity) representing increasing brightness. Whenever luminosity is at 0, the values of saturation and hue do not matter.

White is the center of the top of the cylinder (L=1, S=0). The center line of the cylinder (S = 0) is a line which connects the center of the black plane (L=0, S=0) with white (L=1, S=0) through a series of gray steps (L from 0 to 1, S=0). Whenever saturation is 0, the value of hue does not matter. The outer edge of the cylinder (S=1) represents fully saturated color.

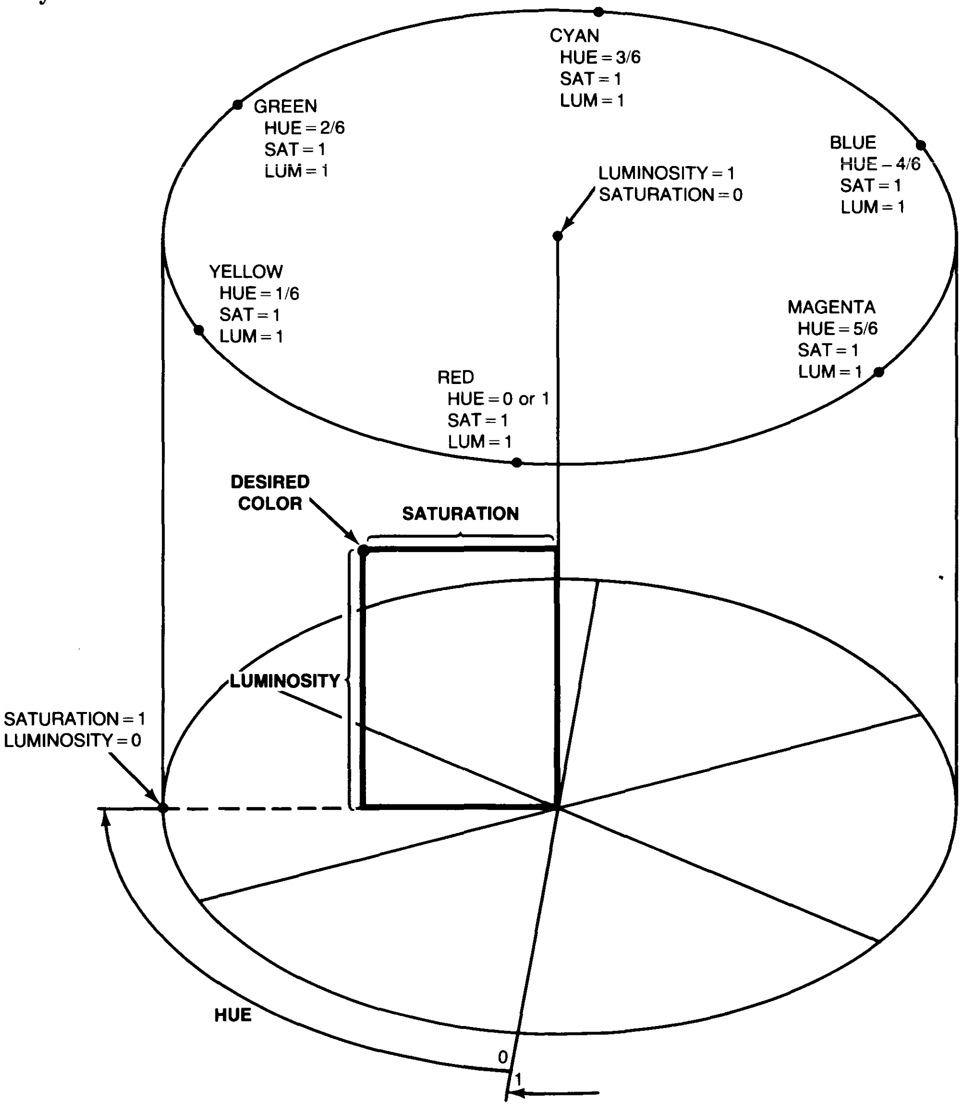


Figure 5-4. HSL Color Specification

Using the above drawing (HSL Color Specification,) hue is the angular coordinate, saturation is the radius, and luminosity is the altitude of the desired color.

HSL to RGB Conversion

Converting from HSL to RGB is simple. Do a SET PEN for the HSL point you want and then read it out of the color map with a GESCAPE. You are limited to the resolution of the color map, but it is very simple. The following line reads the color map into `Old_colors`.

```
14680   GESCAPE 3,2;Old_colors(*)
```

RGB to HSL conversion is not described, due to the fact that it is a one-to-many conversion (the entire bottom plane of the HSL color space is represented by a single point in the RGB color space, and hue is indeterminate if saturation equals 0).

Color Gamuts

The range of colors a physical system can represent is called its *color gamut.* Color gamuts are important when you want to convert between different physical systems, because the source system may be able to produce colors the destination system cannot reproduce. An exhaustive treatment of color gamuts is beyond the scope of this book. However, here are some rules of thumb:

- The color gamuts for CRTs and photographic film are not the same, but are fairly close. If you are lucky, you can photograph the CRT and catch it on film. It may take more than one exposure, so be careful and bracket everything with several exposures.
- The color gamut for printing is significantly smaller than that of either photographic film or of a CRT. The fact that you have a picture of a CRT does not mean you can hand it to a printer and get a faithful reproduction of it.
- The color gamut of a plotter is much smaller than that of a CRT. You have to create images with the limitations of a plotter in mind if you intend to reproduce them on a plotter (see "Plotting and the CRT," below.)

The different color gamuts available are not a problem unless you forget the differences and try to act like all physical systems have the same gamut. Think ahead if you have to reproduce images—it will save a lot a trouble.

Color Hard Copy

It may seem strange to find "Color Hard Copy" a topic under "Color Spaces." The reason it is here is that color hard copy represents a translation between color systems, and many of the problems in color hard copy arise from the fact that the color gamuts available to the CRT and the hard copy device are different.

There are three basic ways to get a color hard copy of what is displayed on a color computer:

- Take a picture of the CRT.
- Re-run the program that generated the image with an external plotter selected as the display device (`PLOTTER IS 705,"HPGL"`).
- Use the PaintJet(TM) Utility.

The first method is the easiest and can capture (usually) whatever is on the CRT, regardless of what colors are used (see "Color Gamuts," above.) The second requires setting up the color map to match the pens in a plotter, and is not as likely to capture what you see on the screen. The third method requires the PaintJet(TM) color printer and the GDUMP_C Utility (on the Utilities 2 Disc). See the "BASIC Utilities Library" section of the *Istalling and Maintaining the BASIC System* manual for information about GDUMP_C.

Photographing the CRT

Photography is an art, not a science. Capturing images off a CRT is relatively straightforward, but sometimes unpredictable due to the different color gamuts available for film and the CRT. The following guidelines will provide a starting point. If your images are not "typical" (whatever that means) you may have to go back and re-photograph some of them. All the CRT images in this manual were captured using these guidelines.

- Use ISO 64 Color film. (The color photos in this manual were taken on Kodak Ektachrome 64.)
- Set up your equipment in a room that can be darkened. It will have to be darkened for the one-second exposure.
- Use a telephoto lens (the longer the better, up to about 500mm). This minimizes the effects of the curvature of the CRT.
- Use a tripod.
- Darken the room and take a one-second exposure.
- Bracket the aperture around f5.6. (One stop above and below.)

Plotting and the CRT

There are two basic reasons the CRT is hard to capture on a plotter.

- The CRT is an additive color device and a plotter is a subtractive color device.
- The color gamut of the CRT is much larger than that of the plotter.

The conversion from additive to subtractive colors is not a huge problem if the plot is a simple line drawing with few intersections and area fills. If the plot is complex, especially with lots of intersections and overlapping filled areas, the plot is much less likely to capture the display image accurately.

A possible technique described below *purposely* limits the color gamut of the CRT to give the plotter some chance of capturing it.

To set up the color map and plotter to match one another:

- Set your background to white (`SET PEN 0 INTENSITY 1,1,1`).
- Set up pens matching the color map colors in slots 1 through 8 in the same order they are presented in the default color map listed under "Default Colors."
- Use pen selectors from 8 through 15 to select your pens.
- Run the program with the color mapped CRT as the display device, modifying it as necessary to produce the image you want on the CRT.
- Re-run the program with the plotter as the display device. You will need to subtract 8 from the pens to properly select the set available on the plotter.

While it is possible to get some idea of the plot that will be produced on the plotter, don't be surprised if they don't look exactly the same. Colors on a CRT are different in source and form from colors on a plotter, as described under "Seeing Color," earlier.

Color References

The following references deal with color and vision. Texts that serve as useful introductions to the topic are starred.

* Cornsweet, T., *Visual Perception.* New York: Academic Press, 1970

Farrell, R. J. and Booth, J. M., *Design Handbook for Imagery Interpretation Equipment* (AD/A-025453) Seattle: Boeing Aerospace Co., 1975

Graham, C. H., (Ed.) *Vision and Visual Perception* New York: J. Wiley & sons, Inc., 1965

* Hurvich, L. M., *Color Vision: An introduction.* Sunderland, MA: Sinauer Assoc., 1980

Judd, D. B., *Contributions to Color Science* (Edited by D. MacAdam; 545) NBS special publication Washington: U. S. Government Printing Office, 1979

* Rose, A., *Vision: human and electronic.* New York: Plenum, 1973

Data Display and Transformations 6

Data Display and Transformations 6

In this chapter, various more advanced topics will be briefly discussed. You are encouraged to load these example routines and try them out after reading the discussion. No program listings will be provided, but the programs/subprograms are on the *Manual Examples Disc* which came with your BASIC software. Every file has at least some code which is general-purpose enough that you can copy program segments into your own applications. The files which are programs can be loaded with a LOAD command. The files which just contain subprograms which can be bodily moved into an application program are in ASCII format; they must be gotten with a GET command. Some of the following routines will work on either monochromatic (black-and-white) or color CRTs, but a few will only work on a color computer. These will be noted as such.

There are several external routines which are called by the following subprograms. They are short, convenient utility subprograms. Listings of these and other utility routines are provided in the "Utility Routines" chapter.

Note that the subprograms stored on the *Manual Examples Disc* and the utility subprograms provided in the "Utility Routines" chapter were included for your convenience. You would need to create applications programs (files of type PROG) to use them.

Bar Charts and Pie Charts

The bar chart routine, which may plot on either a CRT or a plotter, is a general purpose routine.

Below are two sample (random) outputs from the bar chart program. The first shows a "comparative" bar chart; that is, a bar chart in which comparisons between individual bars may easily be made. The second shows a "stacked" bar chart; that is, a bar chart in which bars from the same group are stacked one on top of the other, so that the sums of the bars in each group may be compared.

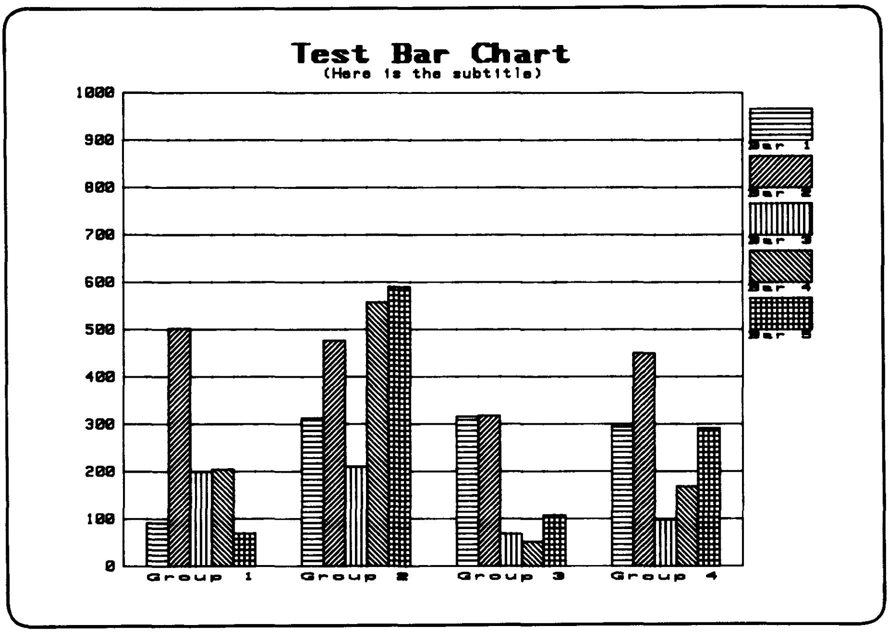


Figure 6-1. Typical Bar Charts

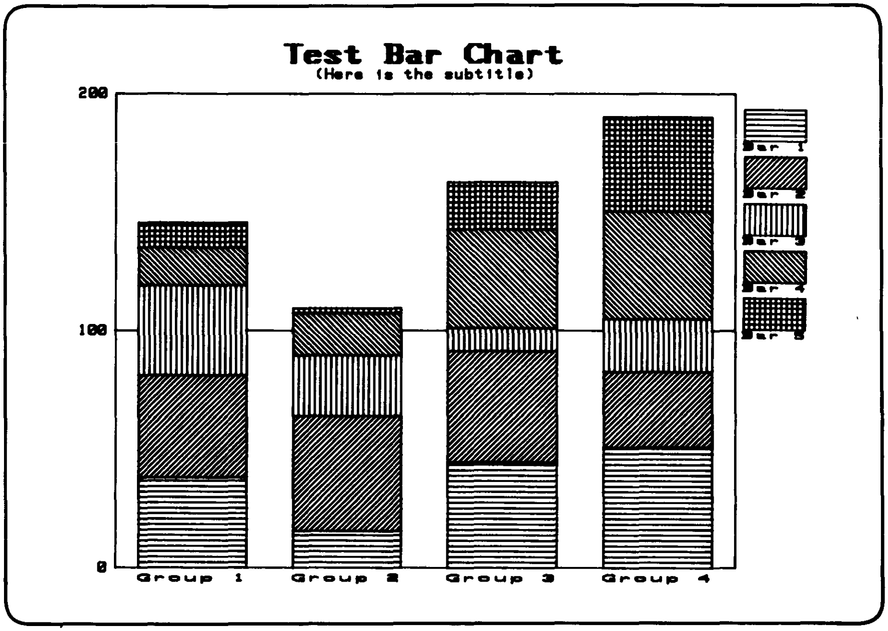


Figure 6-1. Typical Bar Charts (cont.)

The pie-chart subprogram (on the *Manual Examples Disc*) can use both the color map and area fills. This program may be loaded from the *Manual Examples Disc* from the file named `Pie_Chart`. The program sends random data to the subprogram.

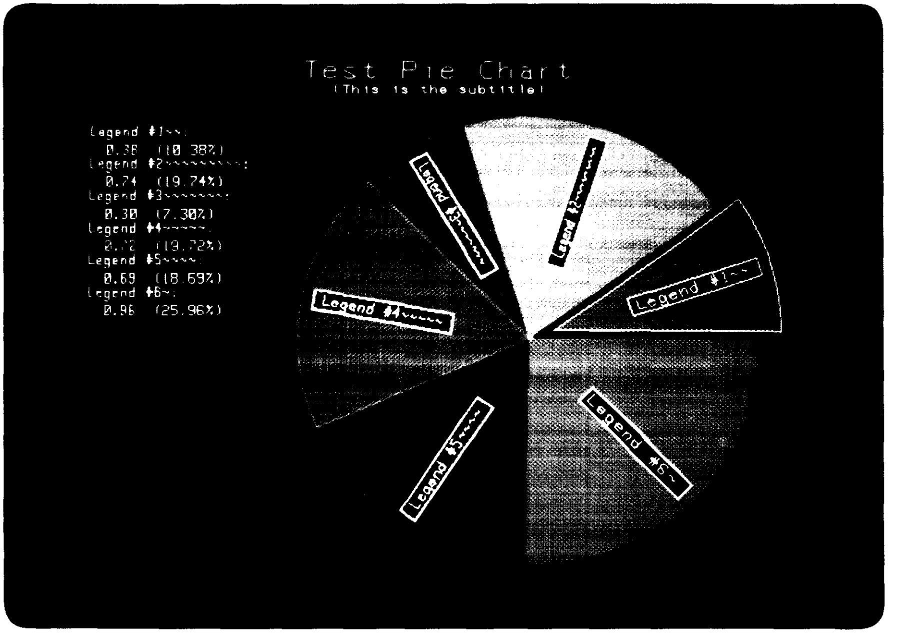


Figure 6-2. Output of "Pie_Chart" Program

Study the program as you require.

Two-Dimensional Transformations

When you want a two-dimensional figure to be drawn after having been scaled, translated, rotated, or sheared, you need to know about the generalized 2D transformations. The purpose of this manual is not to go into theoretical discussions in depth, but some excellent sources will be cited[1].

For 2D graphics, there needs to be a three-column data array: the first two columns are the X and Y coordinates, and the third column is something necessary to keep the mathematics working correctly (refer to the cited works for further discussion).

The transformation matrices for scaling, translation, rotation, and shearing are defined as follows. They all start out as an identity matrix and are modified thus:

2D Scaling Transformation Matrix

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Sx is the scaling factor in the X direction, and Sy is the scaling factor in the Y direction. This means that you can stretch or compress the image along both axes independently.

2D Translation Transformation Matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix}$$

Tx and Ty are the translation factors in the X and Y directions, respectively. Translation (moving the image) can take place in the X and Y directions independently.

---

[1] For an in-depth discussion into many areas of computer graphics, we recommend these books:

Principles of Interactive Computer Graphics, William M. Newman and Robert F. Sproull, 2nd Edition, McGraw-Hill, 1979.

Fundamentals of Interactive Computer Graphics, J. D. Foley and A. Van Dam, Addison-Wesley, 1982.

Mathematical Elements for Computer Graphics, David F. Rogers and J. Alan Adams, McGraw-Hill, 1976.

Seeing: Illusion, Brain, and Mind, John P. Frisby, Oxford University Press.

2D Rotation Transformation Matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This allows you to rotate the image about the origin. θ is the angular distance through which the object is to be rotated, and it is expressed in current units. If you want to rotate the object about some other point than the origin, you must translate that point to the origin, do the rotation, and translate it back to the original point.

2D Shearing Transformation Matrix

$$\begin{pmatrix} 1 & Sh_y & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Shearing is translating different parts of the image different amounts, depending on the value in the other axis. For example, if your data array is the outline of a capital "R", shearing in the X direction with a positive value would "italicize" it; that is, shift the top of the letter farther to the right than the middle of the letter. It would become slanted.

These transformations are applied to the data array by a matrix multiplication (see the MAT statement in the *BASIC Language Reference* manual). To see these operations in action, load the program `Lem2D` from the *Manual Examples* disc.

The different transformations are selected by pressing "T" for translation, "R" for rotation, "S" for scaling, and "H" for shearing. Rotating the knob controls the values put into the transformation matrix. Study the program and accomodate techniques to your system and situation.

Three-Dimensional Transformations

In a logical extension of the two-dimensional transformations, the three dimensional transformations have four columns. Again, this allows the matrix multiplies to work.

3D Scaling Transformation Matrix

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Translation Transformation Matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

When rotating in three dimensions, there are three different axes about which rotation can occur. When rotating points about the X-axis, the Y and Z coordinates of the points change, but not the X coordinates. When rotating about the Y-axis, X and Z coordinates change, but not Y coordinates. When rotating about the Z-axis, X and Y coordinates change, but not Z coordinates. These characteristics become apparent after seeing how the rotation matrices are constructed.

3D Rotation Transformation Matrices

Rotation about X-axis

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about Y-axis

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about Z-axis

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Again, in rotation about an axis in three dimensions, the values in that axis are not changed, only the values in the other two axes are changed. For example, in rotation about the first axis (the X-axis), the first row and first column of the matrix are straight from the identity matrix and therefore do not cause a change in the X-values of the resultant matrix.

3D Shearing Transformation Matrix

$$\begin{pmatrix} 1 & S_{yx} & S_{zx} & 0 \\ S_{xy} & 1 & S_{zy} & 0 \\ S_{xz} & S_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This shearing transformation is a little bit more tricky. Sxy is the shear in the X direction which is proportional to Y, and Sxz is the shear in the X direction which is proportional to Z. The other values work in a similar manner. As you can see, with 3D shearing, the amount of shear is dependent upon the values in *both* the other dimensions.

Surface Plotting

There are three different methods included on the *Manual Examples Disc* for plotting a surface; that is, plotting a two-dimensional array the value of whose elements represent the third dimension at that point. Each method will display the same data so that you can get a feel for the advantages and disadvantages of each method of display. The data, a 100×100 array, is random "mountains" and "valleys" and looks somewhat like old hills worn smooth by erosion.

Contour Plotting

A contour map is a display of a surface from directly above the surface, from an infinite distance. "Infinite" in this context merely means that no perspective effects are included.

The subprogram is passed the surface array, the minimum and maximum contour levels, the contour interval, and three logical variables. These specify

1. whether or not you want the local minima and maxima noted on the output,
2. whether or not you want two lines of "stats"; informational lines concerning array size and contour intervals, and
3. whether or not the plot is to be sent to a CRT. For more information, see the file `Contour` on the *Manual Examples Disc.*

Both the following plots were made with this subprogram. Only the contour interval was changed between the first and second plots. The subprogram was instructed to note the local highs and lows, and also to print the array information at the bottom of the plot.

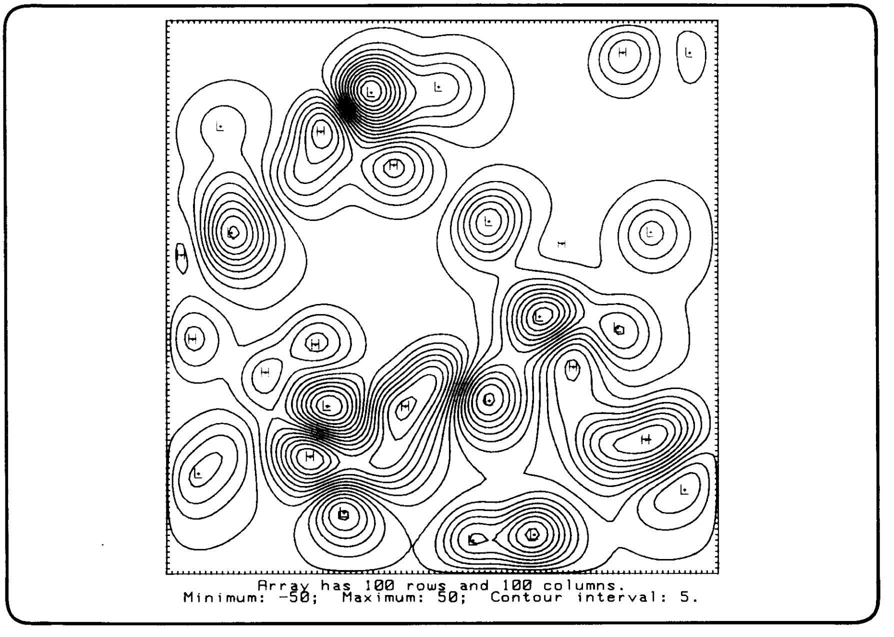


Figure 6-3. Output of "Contour" Subprogram

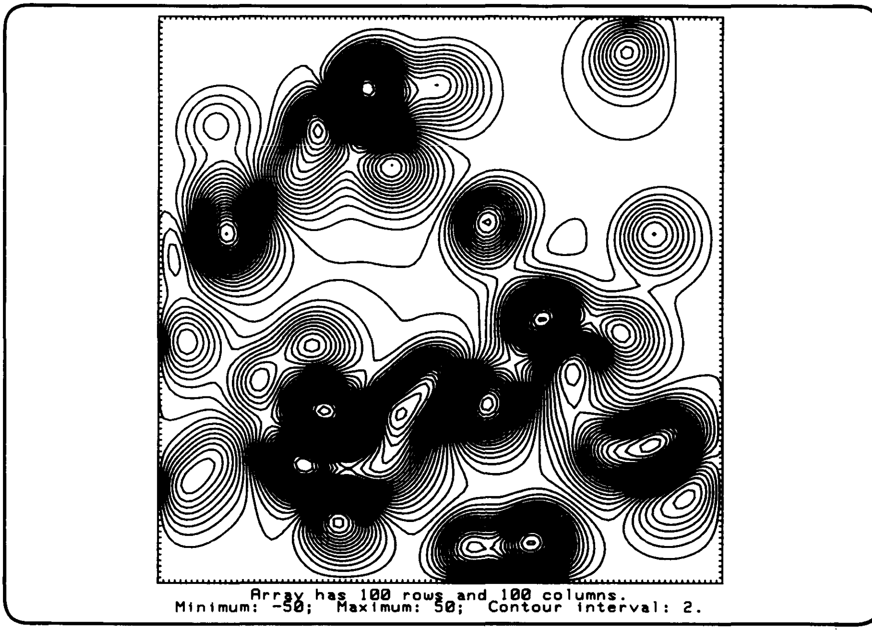


Figure 6-4. More "Contour" Output

Gray Maps

This concept goes back to the days before graphics output devices were in widespread use, and line printers were called upon to plot pictures. Basically, the darkness of a character printed by the printer was proportional to the range in which an element in the array fell. The darkness was caused by overstriking characters in various combinations to produce different amounts of black ink on the page.

The same concept can be used with graphics output devices. The output looks better, of course, because of the increased resolution of graphics output devices over line printers, but the overall result is similar. A gray map can be output to a monochrome or color CRT, and both kinds are presented here. First, the monochrome version. The probability of a pixel being turned on is proportional to the value of the array at that point. To make computation easier, the routine scales the array such that the lowest point becomes zero, and the highest point becomes one. Therefore, the light areas are the high points, the darker areas are the low points, and the average brightness of an area on the screen is proportional to the value in the array at that point.

The program is called `Gray_Map` on the *Manual Examples Disc.*

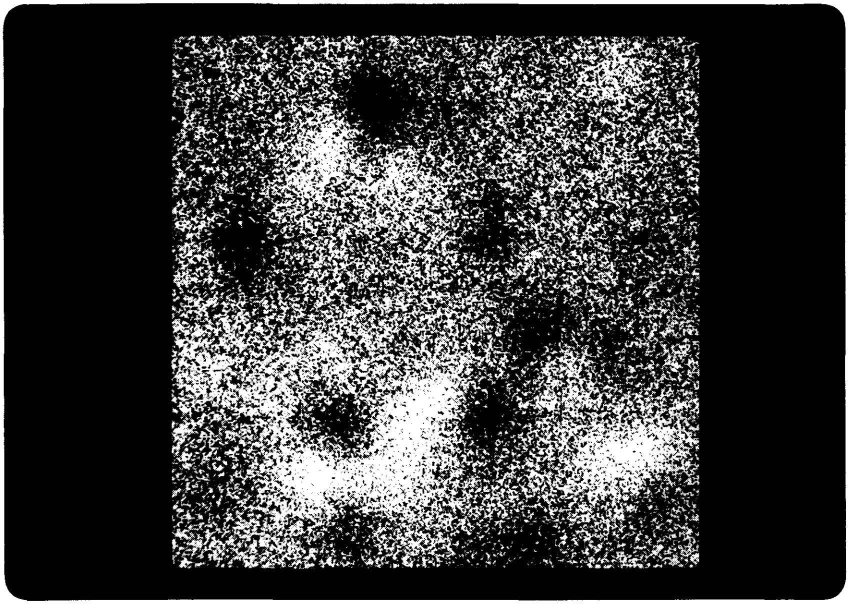


Figure 6-5. Output of "Gray_Map" Program

Next is a Gray Map as drawn on a color-mapped display. It must be a color-mapped display (and not an external color monitor interfaced with an HP 98627A) because the color-map capabilities are needed. The main difference is that instead of the probability of a pixel being turned on being dependent on the array value, all pixels are turned on, and it's the *color* of each pixel which is dependent on the array value.

In Figure 6-6, Pen 0 is not redefined, as it is the background color, but pens 1 through 15 are defined to be varying shades of gray:

```
FOR Pen=1 TO 15
  SET PEN Pen COLOR 0,0,(Pen-1)/14
NEXT Pen
```

Figure 6-7 makes the difference between the highs and the lows more obvious:

```
FOR Pen=1 TO 15
  SET PEN Pen COLOR 2/3+1/3*(Pen>8),ABS(9-Pen),.7
NEXT Pen
```

This will cause the levels *below* the main level to be shades of blue (hue=2/3) and shades above the main level to be shades of red (hue=2/3+1/3=1.0).

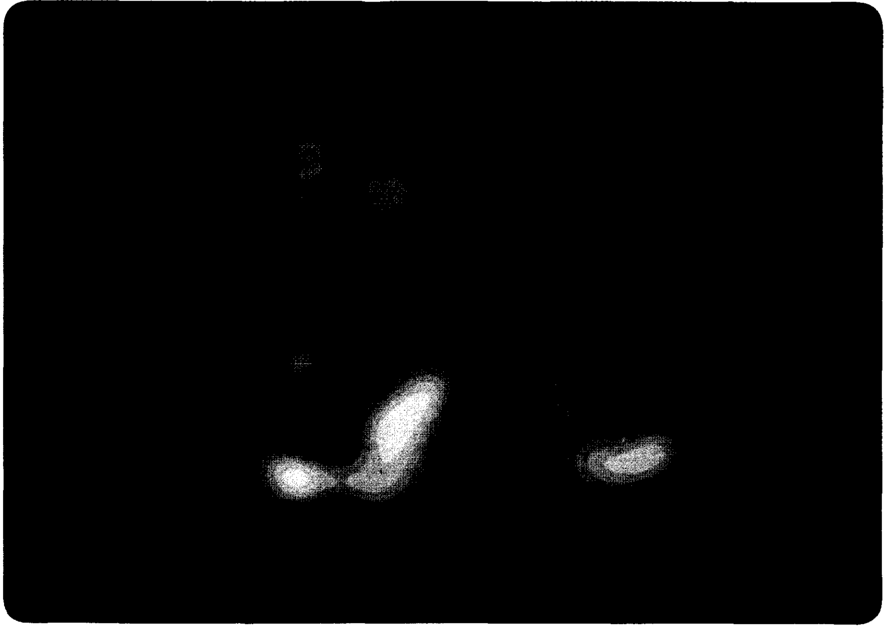


Figure 6-6. "Gray_Map" on a Color-Mapped Display

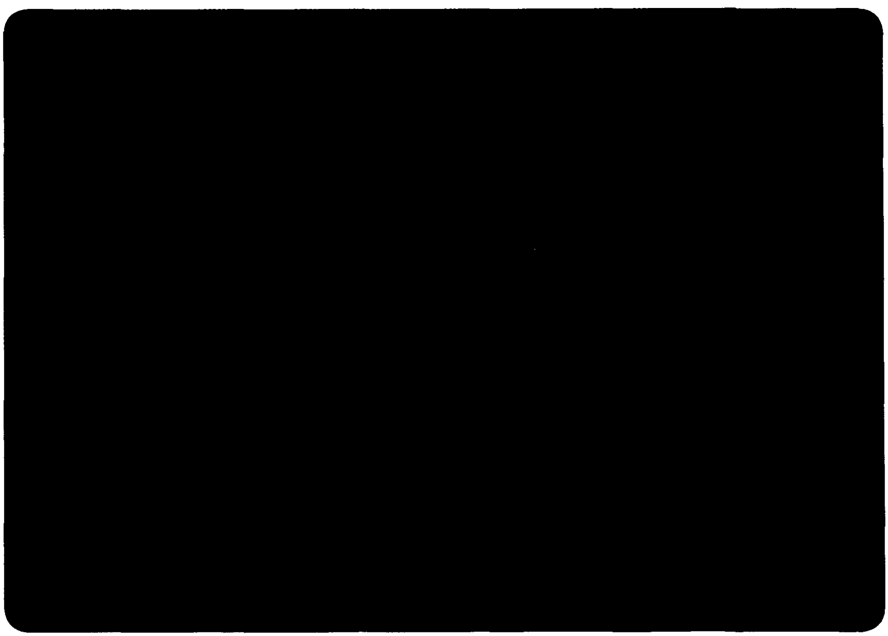


Figure 6-7. "Gray_Map" with Varying Hues

Surface Plot

Another way to look at an array is to look at it from some other angle than straight above. The following routine allows you to look at the surface from above or below. Again, this is the same data as before; notice that the highs and lows are in the same places.

This routine (found on the file `Surface` on the *Manual Examples Disc*) functions by plotting each row of the array as one line on the plotting device. The points of each line are defined to be an offset (determined by which row is currently being plotted and the "height" from which you are looking at the surface) plus the value of the array element you're on. A height array is maintained, the first row of which is the highest point encountered thus far for that column number, and the second row contains the lowest points encountered thus far. If a point is higher than the highest point seen so far, it is visible, and then it becomes the new highest point. The low points are similarly maintained.

The parameter `Front_edge` and `Back_edge` are the height, in GDUs, that the front edge and the back edge of the array are to be from the bottom of the plotting surface. If `Front_edge` is less than `Back_edge`, more of the top surface will be visible. Conversely, if `Front_edge` is greater than `Back_edge`, more of the bottom surface will show.

In the first of the three plots, the variable **Opaque** is passed to the subprogram with a value of 0 (false). Therefore, the surface is treated as if it were transparent, and no hidden lines are removed. This makes the surface hard to interpret because you cannot tell which surface is supposed to be closer to you; *everything* is visible. In the next two plots, **Opaque** is 1 (true), and hidden lines are removed. In the first of the two opaque surfaces, the top is more visible; in the second, the bottom is more visible.

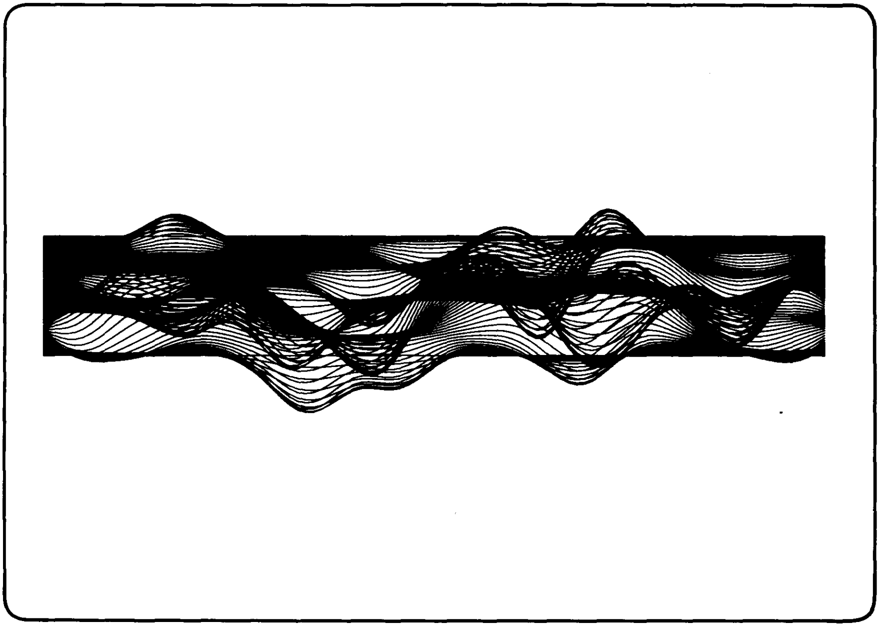


Figure 6-8. "Surface" Program's Output

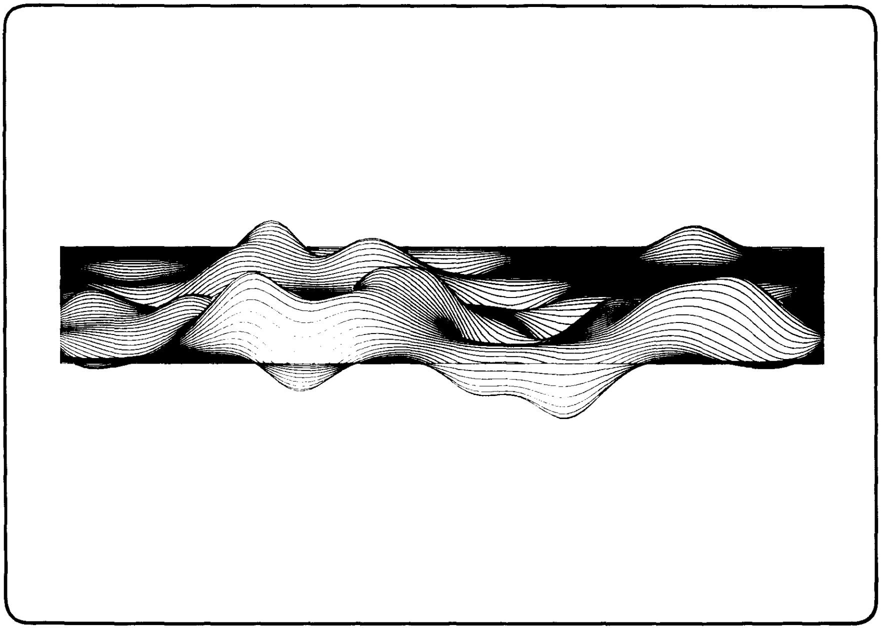


Figure 6-9. "Surface" with Opaque=1 (Top Visible)

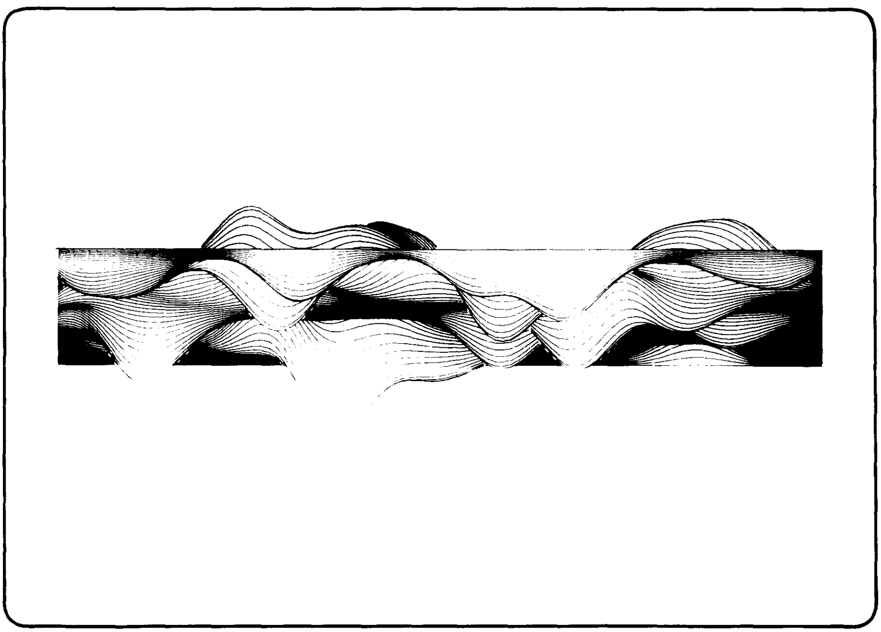


Figure 6-10. "Surface" with Opaque=1 (Bottom Visible)

Utility Routines 7

Utility Routines 7

This chapter consists of several utility routines which are called by some of the subprograms in the "Data Display and Transformation" chapter. Others are included which would be convenient for many graphics applications. A small amount of discussion is included before the routine, if it is necessary.

Drawing Arcs

Note that only two parameters are required. Everything from **Radius** on is optional. The two ON...GOTO statements (lines 130 and 200) take care of the number of parameters passed, assigning default values for only those parameters which were not passed by the calling context.

```
10      ! ************************************************************************
20 Arc:     SUB Arc(X,Y,OPTIONAL Radius_,Start_,End_,Intervals_,Penup_,Aspect_)
30      ! This subroutine draws an arc of a circle with the center at X,Y and a
40      ! radius of "Radius".  The arc starts at a position of "Start" degrees
50      ! and ends at "End" degrees and has a total of "Interval" individual
60      ! line segments.  The greater "Intervals" is, the more rounded the arc
70      ! will look, but also the longer the routine will take to finish.  If
80      ! "Penup" is non-zero, the pen will be picked up before the arc is
90      ! started. If not, it will be left down (assuming it was down before).
100     ! Often you want to draw a straight line to the arc you are starting
110     ! to draw. If "Radius" is positive, the arc will proceed counter-
120     ! clockwise; if negative, clockwise.
130     ON 9-NPAR GOTO 140,150,160,170,180,190,200      ! ON <maxparms>+1-NPAR
140     Aspect=Aspect_
150     Penup=Penup_
160     Intervals=Intervals_
170     End=End_
180     Start=Start_
190     Radius=Radius_
200     ON NPAR-1 GOTO 210,220,230,240,250,260,270      ! NPAR+1-<req. parms>
210     Radius=1.
220     Start=0.
230     End=360.
240     Intervals=INT((End-Start)/5.)
250     Penup=1
260     Aspect=1.
270     DEG
280     IF Penup THEN PENUP
290     IF (Radius>0.) AND (End<=Start) THEN End=End+360.
300     IF (Radius<0.) AND (End>=Start) THEN End=End-360.
310     Step=(End-Start)/Intervals
```

```
320    Radius=ABS(Radius)
330    FOR I=Start TO End STEP Step
340      PLOT X+Radius*Aspect*COS(I),Y+Radius*SIN(I)
350    NEXT I
360    SUBEND
```

Simulating Wide Pens

With the next two subprograms, you can draw pictures that will look like your plotter pen is extremely wide. Theoretically, you could specify that your pen is wider than the whole plotting surface, although not much of a picture would result.

```
10      ! *******************************************************************
20 Fat_line:     SUB Fat_line(X1,Y1,X2,Y2,Thickness,Delta)
30      ! This routine makes a line from point X1,Y1 to point X2,Y2 simulating a
40      ! pen whose tip is width "Thickness".  Delta is the approximate (it may
50      ! be tweaked) distance between actual lines.  The smaller delta is, the
60      ! darker and more accurate the simulation will be, but the execution
70      ! time will suffer.
80      DEG
90      Distance=SQR((X2-X1)^2+(Y2-Y1)^2)
100     Angle=FNAtan(Y2-Y1,X2-X1)
110     Cos_angle=COS(Angle)
120     Sin_angle=SIN(Angle)
130     Perp=Angle+90
140     Cos_perp=COS(Perp)
150     Sin_perp=SIN(Perp)
160     Delta=Thickness/INT(Thickness/Delta)
170     Semithick=Thickness/2
180     Direction=1
190     PENUP
200     FOR Y=-Semithick TO Semithick STEP Delta
210       Dx=SQR(Semithick^2-Y^2)
220       IF Direction THEN
230         PLOT X1+Y*Cos_perp-Dx*Cos_angle,Y1+Y*Sin_perp-Dx*Sin_angle
240         PLOT X2+Y*Cos_perp+Dx*Cos_angle,Y2+Y*Sin_perp+Dx*Sin_angle
250       ELSE
260         PLOT X2+Y*Cos_perp+Dx*Cos_angle,Y2+Y*Sin_perp+Dx*Sin_angle
270         PLOT X1+Y*Cos_perp-Dx*Cos_angle,Y1+Y*Sin_perp-Dx*Sin_angle
280       END IF
290       Direction=NOT Direction
300     NEXT Y
310     SUBEND
```

```
10      ! *********************************************************************
20 Fat_arc:   SUB Fat_arc(X,Y,Radius,Theta1,Theta2,Delta_theta,Thickness,Delta)
30      ! This routine makes an arc centered around point X,Y and radius Radius
40      ! going from Theta1 to Theta2 by Delta_theta, simulating a plotter
50      ! pen whose tip is width "Thickness".  Delta is the approximate (it may
60      ! be tweaked) distance between actual lines.  The smaller delta is, the
```

```
70    ! darker and more accurate the simulation will be, but the execution.
80    ! time will suffer.
90    DEG
100   Semithick=Thickness/2
110   Delta=Thickness/INT(Thickness/Delta)-1.E-13
120   Perp1=Theta1+90
130   Cos_perp1=COS(Perp1)
140   Sin_perp1=SIN(Perp1)
150   Perp2=Theta2+90
160   Cos_perp2=COS(Perp2)
170   Sin_perp2=SIN(Perp2)
180   FOR R=Radius-Semithick TO Radius+Semithick STEP Delta
190     Dx=SQR(Semithick^2-(R-Radius)^2)
200     IF Direction THEN
210       PLOT X+R*COS(Theta1)-Dx*Cos_perp1,Y+R*SIN(Theta1)-Dx*Sin_perp1
220       FOR Theta=Theta1 TO Theta2 STEP Delta_theta
230         PLOT X+R*COS(Theta),Y+R*SIN(Theta)
240       NEXT Theta
250       PLOT X+R*COS(Theta2)+Dx*Cos_perp2,Y+R*SIN(Theta2)+Dx*Sin_perp2
260     ELSE
270       PLOT X+R*COS(Theta2)+Dx*Cos_perp2,Y+R*SIN(Theta2)+Dx*Sin_perp2
280       FOR Theta=Theta2 TO Theta1 STEP -Delta_theta
290         PLOT X+R*COS(Theta),Y+R*SIN(Theta)
300       NEXT Theta
310       PLOT X+R*COS(Theta1)-Dx*Cos_perp1,Y+R*SIN(Theta1)-Dx*Sin_perp1
320     END IF
330     Direction=NOT Direction
340   NEXT R
350   SUBEND
```

Housekeeping

The next few subprograms deal with the humdrum housekeeping chores that need to be done to start and/or end a plot.

```
10      ! ********************************************************************
20 Plotter_is:      SUB Plotter_is(Crt)
30      ! This subroutine defines the plotting device to be used.
40      Crt=FNAsk("Do you want the plot on the CRT?","YES")
50      IF Crt THEN
60        GINIT
70        PLOTTER IS CRT,"INTERNAL"
80      ELSE
90        ON TIMEOUT 7,5 GOTO 140
100       GINIT
110       PLOTTER IS 705,"HPGL"
120       OFF TIMEOUT 7
130       SUBEXIT
140       Message("I've tried for 5 seconds to raise select code 7; no answer.
Defaulting to CRT.")
150       OFF TIMEOUT 7
160       GINIT
170       PLOTTER IS CRT,"INTERNAL"
180       Crt=1
190     END IF
200     SUBEND
```

This next routine forces the user to set P1 and P2 (the lower-left and upper-right corners of the plotting surface, respectively) *before* the PLOTTER IS statement is executed. The reason this is necessary is that the PLOTTER IS statement reads P1 and P2, which define the hard-clip limits. Therefore, if they are set after the PLOTTER IS is executed, they will be ignored, and the old values (the ones in effect when the PLOTTER IS was executed) will be used.

```
10      ! ********************************************************************
20 Load_paper:      SUB Load_paper(OPTIONAL Orientation_$)
30      ! This prompts the user to put the paper in the plotter in the
40      ! orientation, and to define the corners of the paper, BEFORE the
50      ! PLOTTER IS statement is executed.
60      IF NPAR=0 THEN
70        Orientation$=""
80      ELSE
90        Orientation$=Orientation_$
100     END IF
110     SELECT UPC$(TRIM$(Orientation$))
120       CASE "H"
130         Orient$=" horizontally"
140       CASE "V"
150         Orient$=" vertically"
160       CASE ELSE
170         Orient$=""
180     END SELECT
190     BEEP
200     DISP "Put the paper in the plotter";Orient$;", define the corners, and hi
t 'CONT'."
210     PAUSE
220     DISP
230     GINIT
240     PLOTTER IS 705,"HPGL"
250     SUBEND
```

```
10     ! ********************************************************************
20 Gdu:     SUB Gdu(X_gdu_max,Y_gdu_max,OPTIONAL Gdu_xmid,Gdu_ymid)
30     ! This returns Xright, Yhigh and their respective midpoints in GDUs.
40     ! Note that if Gdu_xmid is defined, Gdu_ymid must be also.
50     COM /G_units/ Gdu_xmax,Gdu_ymax,Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax,Show
60     IF Gdu_xmax=0 THEN
70       Gdu_xmax=100*MAX(1,RATIO)
80       Gdu_ymax=100*MAX(1,1/RATIO)
90     END IF
100    X_gdu_max=Gdu_xmax
110    Y_gdu_max=Gdu_ymax
120    IF NPAR>2 THEN
130      Gdu_xmid=Gdu_xmax*.5
140      Gdu_ymid=Gdu_ymax*.5
150    END IF
160    SUBEND
```

Note that in the following routine, the ALPHA and GRAPHICS statements have no effect on multi-plane bit-mapped displays unless the alpha and graphics planes have been separated by appropriate definitions of the write-enable masks.

```
10     ! ********************************************************************
20 Pause:     SUB Pause(OPTIONAL Graphics_)
30     ! This indicates that the output is finished, so push 'CONT' to go on.
40     IF NPAR=0 THEN
50       Graphics=0
60     ELSE
70       Graphics=Graphics_
80     END IF
90     IF Graphics THEN
100      BEEP
110      GRAPHICS OFF
120      ALPHA ON
130    END IF
140    DISP "Push 'CONTINUE' when you're ready to go on."
150    IF Graphics THEN
160      WAIT 2
170      ALPHA OFF
180      GRAPHICS ON
190    END IF
200    PAUSE
210    DISP
220    IF Graphics THEN
230      GRAPHICS OFF
240      ALPHA ON
250    END IF
260    SUBEND
```

```
10     ! ***********************************************************************
20 End_plot:      SUB End_plot(Crt,Copy,Device)
30     ! This is just a housekeeping routine that takes care of some sundries
40     ! at the end of a plot.  "Crt" is a logical variable that tells whether
50     ! the plot was done on the CRT or not.  "Copy" is a variable that is
60     ! returned to the calling routine that tells you whether you want
70     ! another copy of the plot on the hard-copy plotter (Note that if Crt is
80     ! true, Copy is forced to be false).  "Device" is the address of the
90     ! DUMP DEVICE.
100    IF Crt THEN
110      CALL Pause(1)
120      Copy=0
130      IF FNAsk("Shall I 'DUMP GRAPHICS'?","NO") THEN
140        Expanded=FNAsk("...'EXPANDED'?","NO")
150        OUTPUT KBD USING "#,K";Device
160        INPUT "Dump device?",Device
170        IF Expanded THEN
180          DUMP DEVICE IS Device
190        ELSE
200          DUMP DEVICE IS Device,EXPANDED
210        END IF
220        DUMP GRAPHICS
230      END IF
240    ELSE
250      PENUP
260      PEN 0
270      CALL Gdu(X_gdu_max,Y_gdu_max)
280      Setgu
290      MOVE X_gdu_max,Y_gdu_max
300      IF Copy THEN
310        Copy=FNAsk("Do you want another copy of the plot?","NO")
320        IF Copy THEN CALL Load_paper
330      END IF
340    END IF
350    SUBEND
```

Program Efficiency

The following subprogram, `Label`, becomes useful only if there are several labels to be plotted which have different character sizes, orientations, label origins, etc. One call of this routine allows you to set all of the parameters dealing with labelling. Thus, in the calling routine, you need only have one line per label, rather than a CSIZE, LDIR, LORG, PEN, and MOVE for each label.

```
10      ! ********************************************************************
20 Label:     SUB Label(Csize,Asp_ratio,Ldir,Lorg,Pen,X,Y,Text$)
30      ! This defines several systems variables (in CSIZE, LDIR, etc.), and
40      ! labels the text (if any) accordingly.
50      DEG
60      CSIZE Csize,Asp_ratio
70      LDIR Ldir
80      LORG Lorg
90      PEN Pen
100     MOVE X,Y
110     IF Text$<>"" THEN LABEL USING "#,K";Text$
120     PENUP
130     SUBEND
```

The next routine returns the arc tangent *in the correct quadrant* of Y/X, both of which are passed in. If X=0, the routine takes care of it; it doesn't attempt a divide by zero.

```
10      ! *********************************************************************
20 Atan:      DEF FNAtan(Y,X)
30      ! This figures the arc of Y/X in the correct quadrant and takes
40      ! care of multiples of 90 degrees where X=0.  The value returned is in
50      ! current units.
60      Radians=(ACS(-1)=PI)
70      DEG
80      IF X=0 THEN
90        Arctan=(90+180*(Y<0))*(Y<>0)  ! If X=0 and Y=0, Arctan=0.
100     ELSE
110       Arctan=ATN(Y/X)+180*(X<0)+360*((X>0) AND (Y<0))
120     END IF
130     IF Radians THEN
140       RAD
150       Arctan=Arctan/57.2957795131
160     END IF
170     RETURN Arctan
180     FNEND
```

This next routine was called by the Gray Map routine in the *Data Display and Transformations* chapter. It takes an array and re-scales it to fit a new minimum and maximum.

```
10      ! *******************************************************************
20 Scale:      SUB Scale(Surface(*),New_min,New_max)
30      ! This routine scales a matrix such that it will have a new lowest
40      ! value of New_min and a new highest value of New_max.
50      DISP USING "K";"Scaling the surface array from ",New_min," to
",New_max,"."
60      Min=MIN(Surface(*))
70      Max=MAX(Surface(*))
80      IF Min=Max THEN  ! Array is completely flat
90        MAT Surface= (New_min)
100       SUBEXIT
110     END IF
120     MAT Surface= Surface-(Min)
130     Range_recip=(New_max-New_min)/(Max-Min)
140     MAT Surface= Surface*(Range_recip)
150     MAT Surface= Surface+(New_min)
160     DISP
170     SUBEND
```

9845 Graphics System Compatibility

The HP 9845 graphics system allowed the user to go between UDUs and GDUs at will, merely by executing the statements SETUU and SETGU. Series 200/300 BASIC does not have these statements, but they can be simulated by the following short subprograms. (See also subprogram `Gdu` in the "Housekeeping" section, above. It can set the Xmax and Ymax in GDUs.)

```
10      ! **********************************************************************
20 Setgu:      SUB Setgu
30      ! This simulates the 9845 graphics statement SETGU.
40      COM /G_units/ Gdu_xmax,Gdu_ymax,Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax,Show
50      WINDOW 0,Gdu_xmax,0,Gdu_ymax
60      SUBEND
```

```
10      ! **********************************************************************
20 Setuu:      SUB Setuu
30      ! This simulates the 9845 graphics statement SETUU.
40      COM /G_units/ Gdu_xmax,Gdu_ymax,Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax,Show
50      IF Show THEN
60        SHOW Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax
70      ELSE
80        WINDOW Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax
90      END IF
100     SUBEND
```

```
10      ! **********************************************************************
20 Show:       SUB Show(Xleft,Xright,Ylow,Yhigh)
30      ! This simulates the system command SHOW, but saves the variables so
40      ! the routines Setgu and Setuu work.
50      COM /G_units/ Gdu_xmax,Gdu_ymax,Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax,Show
60      IF Gdu_xmax=0 THEN
70        Gdu_xmax=100*MAX(1,RATIO)
80        Gdu_ymax=100*MAX(1,1/RATIO)
90      END IF
100     Udu_xmin=Xleft
110     Udu_xmax=Xright
120     Udu_ymin=Ylow
130     Udu_ymax=Yhigh
140     Show=1
150     SHOW Xleft,Xright,Ylow,Yhigh
160     SUBEND
```

```
10      ! ********************************************************************
20 Window:      SUB Window(Xleft,Xright,Ylow,Yhigh)
30      ! This simulates the system command WINDOW, but saves the variables so
40      ! the routines Setgu and Setuu work.
50      COM /G_units/ Gdu_xmax,Gdu_ymax,Udu_xmin,Udu_xmax,Udu_ymin,Udu_ymax,Show
60      IF Gdu_xmax=0 THEN
70        Gdu_xmax=100*MAX(1,RATIO)
80        Gdu_ymax=100*MAX(1,1/RATIO)
90      END IF
100     Udu_xmin=Xleft
110     Udu_xmax=Xright
120     Udu_ymin=Ylow
130     Udu_ymax=Yhigh
140     Show=0
150     WINDOW Xleft,Xright,Ylow,Yhigh
160     SUBEND
```

HPGL

The following subprogram specifies the maximum speed at which a plotter should draw. This was made specifically for an HP 9872 plotter, which has a maximum pen speed of 36 cm/sec. If your plotter has a different maximum speed, you will need to change line 100 to reflect the new maximum speed.

```
10      ! ********************************************************************
20 Pen_speed:      SUB Pen_speed(Speed,OPTIONAL Device_)
30      ! This sends an HPGL plotter the command to draw at a maximum speed.
40      IF NPAR=1 THEN
50        Device=705
60      ELSE
70        Device=Device_
80      END IF
90      IF Speed=0 THEN INPUT "What should the maximum plotter speed?",Speed
100     Speed=MIN(MAX(1,INT(Speed+.5)),36)
110     OUTPUT Device USING "#,K";"VS"&VAL$(Speed)&";"
120     SUBEND
```

Miscellaneous

The next two subprograms are not explicitly graphics routines, but they are very useful general-purpose routines and they are used both in previous routines in this chapter, and in the large programs of the *Data Display and Transformations* chapter.

```
10      ! *********************************************************************
20 Ask:      DEF FNAsk(Question$,Default$,OPTIONAL Timeout)
30      ! This is a Yes-or-no question-answering function.  The question is
40      ! in to the function, asked of the user, and the default answer can be
50      ! accepted.  If the user answers intelligibly, that answer is returned
60      ! through the function name; 1 for yes, and 0 for no.  If the user
70      ! responds unintelligibly, the computer beeps, draws attention to the
80      ! fact that an illegal answer was given, re-asks the question, and will
90      ! again accept the default answer.
100     ! If Timeout is passed the question will be asked for that specified
110     ! number of seconds before the default answer is assumed.  If Timeout is
120     ! not passed, it will wait indefinitely for user response.
130     DIM Answer$[160]
140     IF NPAR=3 THEN
150       ON DELAY Timeout GOTO Take_default
160       DISP Question$
170       ON KBD ALL GOTO Process_key
180 Spin:     GOTO Spin    ! "at warp 10, we're goin' nowhere mighty fast..."
190 Process_key:    OFF DELAY
200       Key$=KBD$
210       SELECT Key$[1,1]
220         CASE CHR$(255)  ! It was a non-ASCII keypress.....................
230           SELECT Key$[2,2]
240             CASE "E","C"  ! Enter or Continue?............................
250               GOTO Take_default
260             CASE ELSE  ! Illegal non-ASCII key............................
270               BEEP
280           END SELECT  ! (select key$[2,2])
290         CASE ELSE  ! ASCII keystroke......................................
300           OUTPUT KBD USING "#,K";Key$
310       END SELECT  ! (select key$[1,1])
320       OFF KBD
330     END IF  ! (if npar=3)
340     LOOP ! Now that we're in this loop, we'll stay until we get a good answer
350       DISP Question$;
360       LINPUT "",Answer$
370       Answer$=UPC$(TRIM$(Answer$))
380       IF Answer$="" THEN Answer$=UPC$(TRIM$(Default$))
390 Convert_answer:     SELECT Answer$
400         CASE "YES","Y","1"  ! Affirmative.................................
410           RETURN 1
420         CASE "NO","N","0"  ! Negative.....................................
430           RETURN 0
440         CASE ELSE  ! Huh?!?...............................................
```

```
450         CALL Message("Please answer with a YES or a NO.")
460       END SELECT
470     END LOOP
480 Take_default:    DISP
490     OFF DELAY
500     Answer$=UPC$(TRIM$(Default$))
510     GOTO Convert_answer
520     FNEND

10      ! ******************************************************************
20 Message:    SUB Message(Message$,OPTIONAL Wait_)
30      ! This subroutine displays a message on the DISPlay line of the CRT,
40      ! usually to notify the user of an error, or that a section of code has
50      ! finished executing, etc.  If Wait_ is not defined [passed], the
60      ! computer will beep, and the message will be displayed for two seconds,
70      ! then disappear.  If Wait_ is defined, the computer will beep if it is
80      ! greater than or equal to zero, it will not beep if it is less than
90      ! zero, and in either case, the wait will be the absolute value, rounded
100     ! to the nearest millisecond, unless it is zero, in which case the
110     ! message will not be erased at all.
120     DISP Message$
130     IF NPAR=1 THEN                  ! Default:
140       BEEP
150       WAIT 2                        ! Wait 2 seconds, then
160       DISP                          ! clear the message.
170     ELSE  ! (npar=2)
180       IF Wait_>=0 THEN BEEP         ! Note that the rounding occurs AFTER the
190       Wait=PROUND(ABS(Wait_),-3)    ! BEEP.  This allows "negative zero" which
200       IF Wait>0 THEN                ! not only will not beep, but it will leave
210         WAIT Wait                   ! the message displayed, avoiding the WAIT
220         DISP                        ! and DISP.  A "negative zero" is simulated
230       END IF  ! (if wait>0)         ! by passing a negative number which will
240     END IF  ! (if npar=1)           ! round to zero; e.g., -.0001.
250     SUBEND
```

Appendix A

Appendix A

For your convenience, below is a table and a description of the programs and subprograms on the *Manual Examples Disc*. First is a table of the concepts and capabilities that the various programs exhibit. Following that is an alphabetic listing of the file names with a short description of them.

Table A-1. Example Programs and Their Characteristics

File Name	Color-map/Monochrome/Either	Data Display	Labelling	Interactive	Softkey Control	Knob Control	Data-Driven Plotting/Array Plotting	Erasing Lines	Storing Images	Dithering	Non-Dominant Drawing	Transformation Matrices	User Color Specification	Color Map Definition	Color Map Animation	Attention Getter
SinViewprt	E	•														
Csize	E		•													
CharCell	E		•													
Lorg	E		•													
Ldir	E		•													
SinLabel	E	•	•													
SinLabel2	E	•	•													
SinAxes	E	•	•													
SinGrdAxes	E	•	•													
Pen	E							•								•
Gstore	M	•	•	•		•		•	•							
Lem2	E						•			•						
Rplot	E						•	•		•						
Iplot	E		•				•									
Scenery	E						•			•						
Symbol	E		•				•									
BAR_KNOB	E	•	•	•		•		•								
CIRCLES	C	•	•	•	•						•			•		
BACKGROUND	C	•		•		•	•	•			•	•		•	•	•
MARQUEE	C		•											•	•	•
RIPPLES	C			•		•							•	•	•	•
STORM	C						•							•	•	•
Animation	C													•	•	•
STEREO	C	•		•	•		•				•	•		•		
Pie_Chart	C	•	•													
Lem2D	E	•		•		•	•					•				
Contour[1]	E	•	•				•									
Gray_Map[1]	E	•					•									
Surface[1]	E	•					•									
DumpGraph[1]	M								•							

[1] These are subprograms only, and must be called from a main program. All others are stand-alone programs.

Animation
Any of three scenes can be portrayed as flashing by at high speed; some rushing at you, some rushing away. Demonstrates color map animation.

BACKGROUND
Demonstrates color map definition, non-dominant drawing, three-dimensional transformations, and knob interaction. A box is rotated (repeatedly drawn and erased) in front of a grid without damaging the grid. The display is flicker-free because one image is drawn invisibly while the last image remains. The color map is altered to make the new image visible, while the old, now invisible, image is erased and a new one is drawn.

BAR_KNOB
Demonstrates the use of the knob to control dynamic displays.

CharCell
Shows the relationship between the actual character size and the character cell size.

CIRCLES
This shows that the color map can be defined to simulate an additive color scheme, a subtractive color scheme, or any arbitrary color scheme.

Contour
This *subprogram* accepts a two-dimensional array and plots a contour map. The user may specify low and high contour level and contour interval.

Csize
Demonstrates how to use the CSIZE statement to change the size of the character cells into which labelled characters are placed.

DumpGraph
This *subprogram* takes an image from the frame buffer of a monochromatic CRT and sends it to an HP 82905A printer.

Gray_Map
This *subprogram* accepts a two-dimensional array and plots a gray map from it. The data is scaled from zero to one.

Gstore

Demonstrates the use of GSTORE and GLOAD in quickly replotting the unchanging part of an otherwise dynamic image.

Iplot

Uses incremental plotting to create characters for plotting labels in a user-defined character set.

Ldir

Demonstrates how the LDIR statement allows labelling of text on the graphics screen at any desired angle.

Lem2

Lem2 shows how the pen-control parameter lifts and drops the pen. It takes the same data and plots it in one statement. Uses area fills.

Lem2D

This demonstrates the four basic two-dimensional graphics transformations: translation, rotation, scaling and shearing. The knob controls the values entered, and "T", "R", "S", and "H", respectively, select the operations.

Lorg

Demonstrates how the LORG statement allows centering or cornering of labels in both the X and Y directions.

MARQUEE

Uses color-map animation to create a movie marquee announcing the coming attractions.

Pen

Demonstrates drawing modes on monochromatic CRTs. Lines are drawn, erased and complemented.

Pie_Chart

This program runs a *subprogram* which accepts pie chart data: up to fourteen segments, each with its own label, plus title and subtitle.

RIPPLES
Color map animation with concentric circles. The luminosity of the color represents the height of the ripple on the water.

Rplot
Uses RPLOT statement to move subpictures, PIVOT to rotate them, and AREA INTENSITY to define shading.

Scenery
Uses POLYGONS, POLYLINES, RPLOTS, and area fills to create an idyllic scene of rustic simplicity.

SinAxes
This is part of the "Progressive Example" in Chapters 1 and 2. Axes are added, along with labels at approriate points along them.

SinGrdAxes
This is part of the "Progressive Example" in Chapters 1 and 2. Both a GRID and two AXES statements are used to allow ease of interpolation of values on the data curve and also to avoid clutter.

SinLabel
This is part of the "Progressive Example" in Chapters 1 and 2. Labels are plotted after having used CSIZE, LORG and LDIR.

SinLabel2
This example is similar to SinLabel except that it draws a "bold" label.

SinViewprt
This is part of the "Progressive Example" in Chapters 1 and 2. A viewport is defined using GDU measurements of the screen.

STEREO
Uses non-dominant drawing and three-dimensional transformations to display red-blue stereo images which can be viewed through bi-colored glasses.

STORM

Demonstrates the use and speed of color map animation. A little house on the prairie is besieged by a thunderstorm.

Surface

This *subprogram* draws a surface represented by a two-dimensional array. Hidden lines may be removed, and the viewing angle can be selected by the user.

Symbol

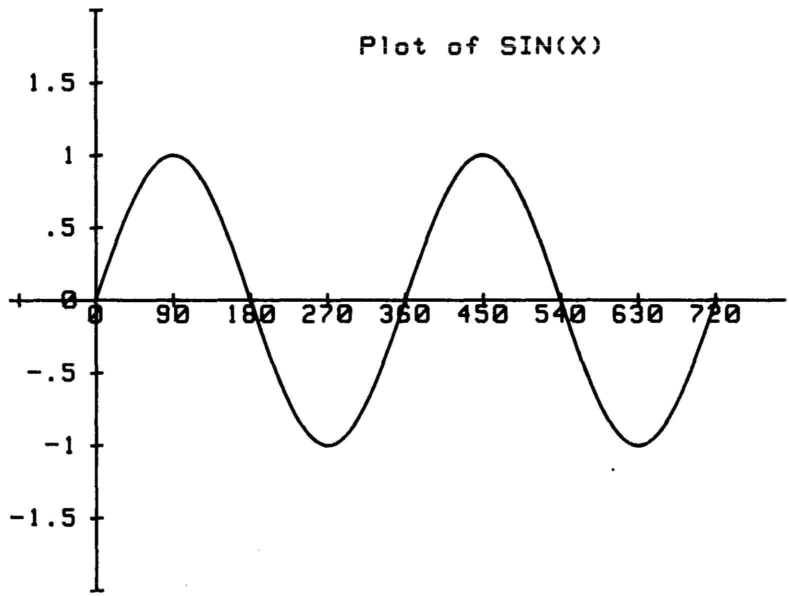
Demonstrates how to define and label user-defined characters with the SYMBOL statement.

Example Graphics Programs

The following programs use graphics to help illustrate the operation of several of the graphics statements available in BASIC. You may wish to modify or entirely rewrite the programs to better understand how the statements work.

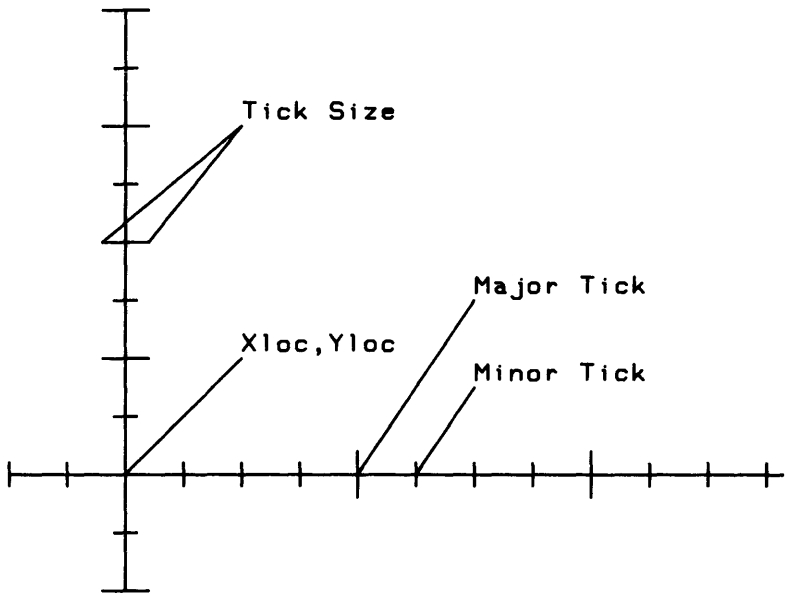
SINE

```
10    ! Program: SINE
20    !
30    ! Shows some basics of drawing and labeling.
40    !
50    DEG                            ! DEGREES
60    GINIT                          ! INITIALIZE
70    GRAPHICS ON                    ! RASTER ON
80    PRINT CHR$(12);                ! CLEAR ALPHA
90    WINDOW -100,800,-2,2           ! SET WINDOW
100   AXES 90,.5                     ! DRAW AXES
110   !
120   LORG 6                         ! LABEL X AXIS
130   FOR I=0 TO 720 STEP 90
140   MOVE I,0
150   LABEL I
160   NEXT I
170   !
180   LORG 8                         ! LABEL Y AXIS
190   FOR I=-1.5 TO 1.5 STEP .5
200   MOVE 0,I
210   LABEL I
220   NEXT I
230   !
240   LORG 5                         ! LABEL PLOT
250   MOVE 450,1.75
260   LABEL "Plot of SIN(X)"
270   !
280   MOVE 0,0                       ! PLOT SINE
290   FOR X=0 TO 720
300   DRAW X,SIN(X)
310   NEXT X
320   !
330   END
```


Plot of SIN(X)
1.5
1
.5
0
-.5
-1
-1.5
0
90
180
270
360
450
540
630
720

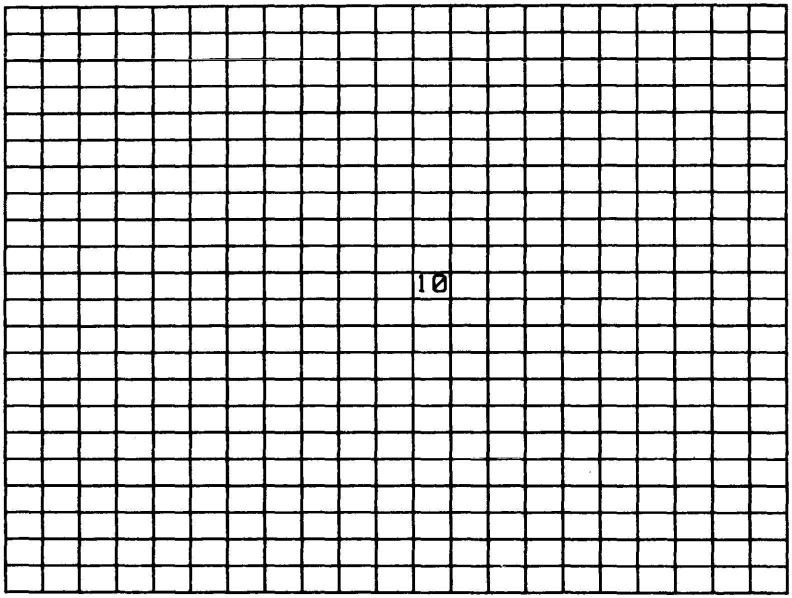
AXES

```
10    ! Program: AXES
20    !
30    ! Draw and label the AXES statement.
40    !
50    GINIT
60    GRAPHICS ON
70    ALPHA OFF
80    !
90    Xloc=20                  ! X AXIS LOCATION
100   Yloc=20                  ! Y AXIS LOCATION
110   Xmaj=4                   ! MAJOR TICK COUNT
120   Ymaj=2                   ! MAJOR TICK COUNT
130   Size=8                   ! LENGTH OF TICKS
140   !
150   FOR I=100 TO 10 STEP -1
160   PEN -1
170   AXES Xtic,Ytic,Xloc,Yloc,Xmaj,Ymaj,Size
180   Xtic=I
190   Ytic=I
200   PEN 1
210   AXES Xtic,Ytic,Xloc,Yloc,Xmaj,Ymaj,Size
220   NEXT I
230   !
240   MOVE Xloc,Yloc           ! LABEL THE AXES
250   IDRAW 20,20
260   LABEL "Xloc,Yloc"
270   MOVE Xloc+40,Yloc
280   IDRAW 20,30
290   LABEL "Major Tick"
300   MOVE Xloc+50,Yloc
310   IDRAW 10,15
320   LABEL "Minor Tick"
330   MOVE Xloc-Size/2,Yloc+40
340   DRAW 40,80
350   MOVE Xloc+Size/2,Yloc+40
360   DRAW 40,80
370   LABEL "Tick Size"
380   !
390   END
```

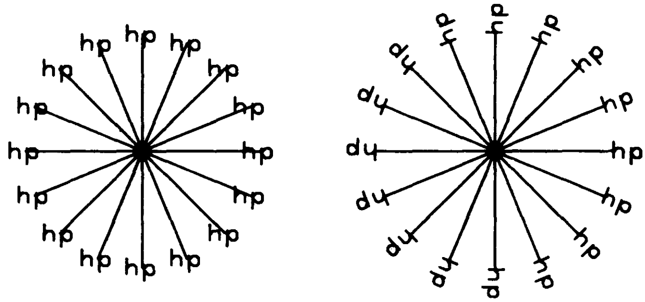

Tick Size
Major Tick
Xloc,Yloc
Minor Tick

GRID

```
10    ! Program: GRID
20    !
30    ! Shows various size grids.
40    !
50    GINIT
60    GRAPHICS ON
70    PRINT CHR$(12);
80    !
90    WINDOW -110,100,-110,110
100   !
110   Yloc=0                ! CENTER AT 0,0
120   Xloc=0
130   Xmaj=6
140   Ymaj=2
150   Size=20
160   !
170   LORG 4
180   !
190   FOR I=10 TO 100 STEP 2
200     Xtic=I
210     Ytic=I
220     GCLEAR
230     MOVE I/2,0
240     LABEL I
250      GRID Xtic,Ytic,Xloc,Yloc,Xmaj,Ymaj,Size
260     WAIT (100-I)/100
270   NEXT I
280   !
290   WAIT 2
300   GRAPHICS OFF
310   END
```


10

LABEL



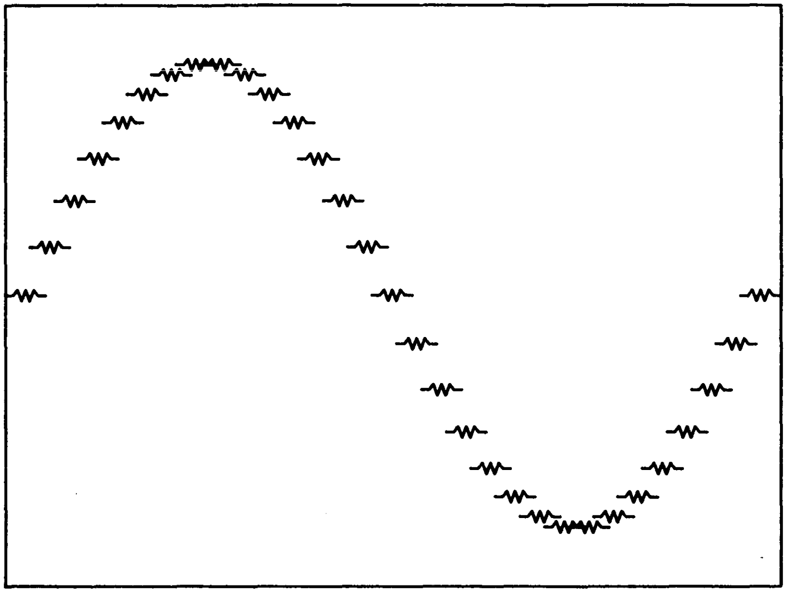
```
10    ! Program: LABEL
20    !
30    DEG
40    GINIT
50    GRAPHICS ON
60    Clear_crt$=CHR$(255)&CHR$(75)
70    OUTPUT 2;Clear_crt$;
80    SHOW -100,100,-100,100
90    !
100   FOR I=0 TO 360 STEP 22.5 ! Non-rotated
110     MOVE -60,0
120     PIVOT I
130     IDRAW 40,0
140     Orgx=40*COS(I)-60
150     Orgy=40*SIN(I)
160     MOVE Orgx,Orgy
170     LORG 5
180     LABEL "hp"
190   NEXT I
200   !
210   FOR I=0 TO 360 STEP 22.5  ! Rotated
220     MOVE 60,0
230     PIVOT I
240     IDRAW 40,0
250     Orgx=40*COS(I)+60
260     Orgy=40*SIN(I)
270     MOVE Orgx,Orgy
280     LORG 2
290     LDIR I                  ! Note LDIR used
300     LABEL "hp"
310   NEXT I
320   !
330   END
```

RevLABEL


```
10    ! Program "RevLABEL".
20    GINIT
30    PLOTTER IS CRT,"INTERNAL"
40    GRAPHICS ON
50    CSIZE 8,-.6    ! Note negative aspect ratio.
60    MOVE 90,50
70    LABEL "Reverse Graphics"
80    END
```

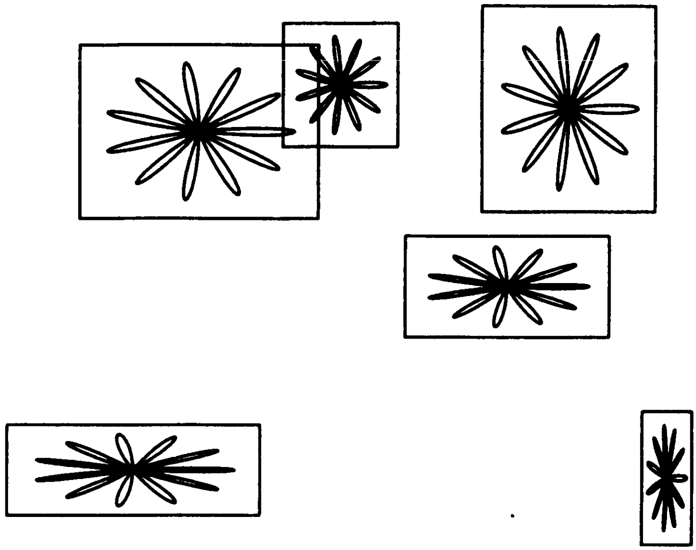
RPLOT

```
10    ! Program: RPLOT
20    !
30    ! Repeats an image at various locations.
40    !
50    DEG
60    GINIT
70    GRAPHICS ON
80    WINDOW -10,370,-100,100
90    PRINT CHR$(12);   ! CLEAR SCREEN
100   DISP " RPLOT"
110   FRAME
120   !
130   FOR I=0 TO 360 STEP 12
140     MOVE I,SIN(I)*80
150     GOSUB Shape
160   NEXT I
170   !
180   GOTO Quit
190   !
200 Shape:   ! DRAW A RESISTOR
210   !
220   RPLOT -10,0,1
230   RPLOT -6,0
240   RPLOT -4,2
250   RPLOT -2,-2
260   RPLOT 0,2
270   RPLOT 2,-2
280   RPLOT 4,2
290   RPLOT 6,0
300   RPLOT 10,0
310   RETURN
320   !
330 Quit:   END
```



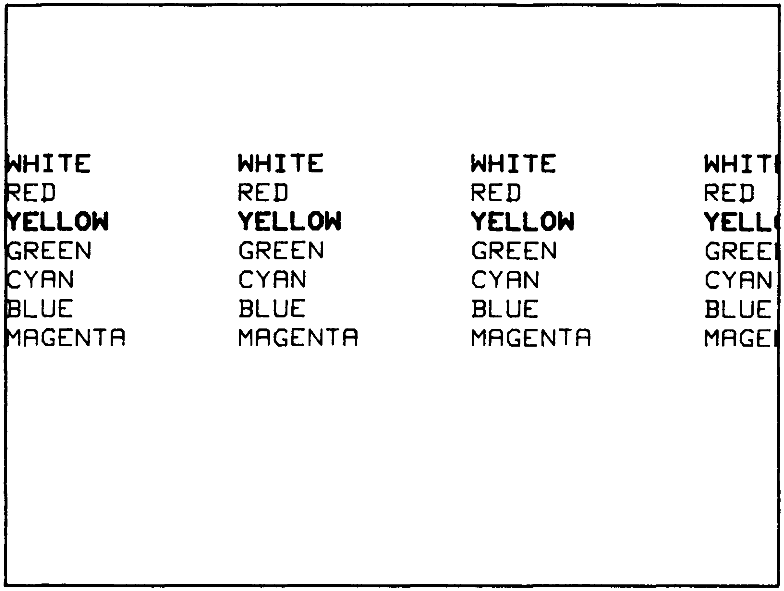
RANDOMVIEW

```
10     ! Program: RANDOMVIEW
20     !
30     RANDOMIZE
40     !
50 Start: ! Demonstration of VIEWPORT and WINDOW
60     !
70     DEG
80     GINIT
90     GRAPHICS ON
100    ALPHA OFF
110    !
120    ! Generate some random numbers
130    !
140    Xmin=RND*131
150    Xmax=Xmin+RND*(131-Xmin)
160    Ymin=RND*100
170    Ymax=Ymin+RND*(100-Ymin)
180    !
190    ! Set VIEWPORT to random area
200    !
210    VIEWPORT Xmin,Xmax,Ymin,Ymax
220    WINDOW -50,50,-50,50
230    FRAME
240    !
250    ! Draw a rose within the area
260    !
270    FOR I=0 TO 200
280      P=40*COS(11*I)          ! ELEVEN LEAF ROSE
290      X=P*COS(I)
300      Y=P*SIN(I)
310      DISP INT(Xmax-Xmin);":";INT(Ymax-Ymin)
320       IF I=0 THEN MOVE X,Y
330      DRAW X,Y
340    NEXT I
350    !
360    GOTO Start               ! DO IT AGAIN
370    END
```

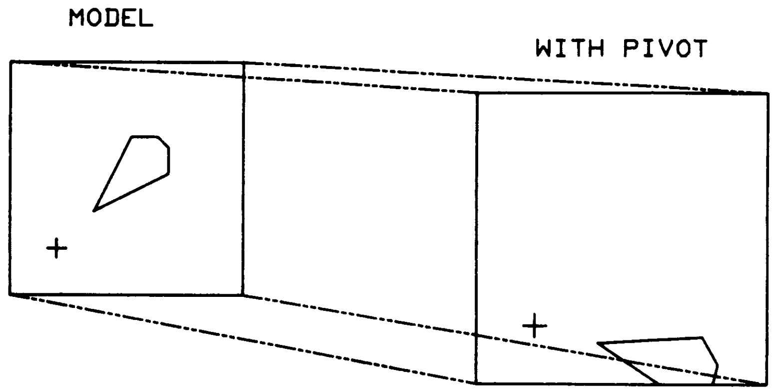


COLOR

```
10    ! Program: COLOR
20    !
30    ! This program works with the 98627A
40    ! Color Output Interface.
50    !
60    !
70    ! Note that a 'PLOTTER IS' statement must
80    ! immediately follow 'GINIT' statement.
90    !
100   ! Note different pen assignments.
110   !
120   GINIT
130   PLOTTER IS 28,"98627A"
140   GRAPHICS ON
150   PEN 1
160   FRAME
170   !
180   FOR X=0 TO 120 STEP 40
190   MOVE X,70
200   PEN 1
210   LABEL "WHITE"
220   PEN 2
230   LABEL "RED"
240   PEN 3
250   LABEL "YELLOW"
260   PEN 4
270   LABEL "GREEN"
280   PEN 5
290   LABEL "CYAN"
300   PEN 6
310   LABEL "BLUE"
320   PEN 7
330   LABEL "MAGENTA"
340   NEXT X
350   END
```


WHITE WHITE WHITE WHIT
RED RED RED RED
YELLOW YELLOW YELLOW YELL
GREEN GREEN GREEN GREE
CYAN CYAN CYAN CYAN
BLUE BLUE BLUE BLUE
MAGENTA MAGENTA MAGENTA MAGE

PIVOT

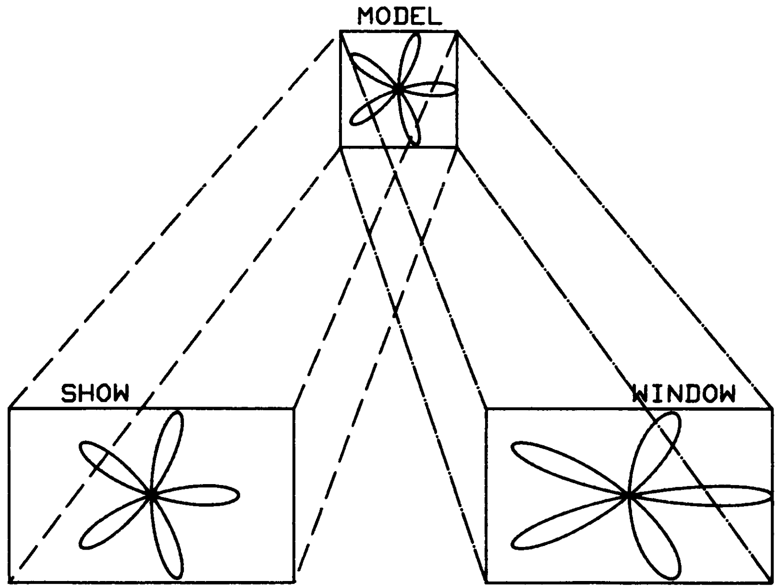



```
10    ! Program: PIVOT
20    !
30    ! Shows pivoting around a point.
40    !
50    DEG
60    GINIT
70    GRAPHICS ON
80    PRINT CHR$(12);
90    DIM X(4),Y(4)
100   DATA 40,20,0,14,-6,6,-14,0,-20,-40 ! SHAPE
110   FOR I=0 TO 4
120     READ X(I),Y(I)
130   NEXT I
140   !
150   DATA 80,130,35,85,0,40,50,90 ! 'WINDOWS'
160   READ Sl,Sr,Sb,St,Ml,Mr,Mb,Mt
170   !
180   DIM Orgx(3),Orgy(3)
190   DATA 40,60,40,40,20,20,0,0 ! ORIGINS
200   FOR I=0 TO 3
210     READ Orgx(I),Orgy(I)
220   NEXT I
230   MOVE 10,95
240   LABEL "MODEL"
250   MOVE 90,90
260   LABEL "WITH PIVOT"
270   LINE TYPE 8
280   MOVE Ml,Mb                    ! CONNECT LINES
290   DRAW Sl,Sb
300   MOVE Mr,Mb
310   DRAW Sr,Sb
320   MOVE Mr,Mt
330   DRAW Sr,St
```

```
340   MOVE S1,St
350   DRAW M1,Mt
360   !
370   MOVE M1,Mt
380   P=1
390   LINE TYPE 1
400   Ox=Orgx(Index)
410   Oy=Orgy(Index)
420   GOSUB Model
430   !
440   VIEWPORT S1,Sr,Sb,St
450   SHOW -25,100,-25,100
460   GOSUB Shape
470   DISP "Angle =";Angle
480   Angle=Angle+5
490   IF Angle<361 THEN 460
500   CALL Cursor(Ox,Oy,-1)         ! PIVOT POINT
510   P=-1
520   GOSUB Model
530   Angle=0
540   Index=Index+1
550   IF Index>3 THEN Quit
560   GOTO 380
570   !
580 Model:  VIEWPORT M1,Mr,Mb,Mt
590         SHOW -25,100,-25,100
600         FRAME
610         GOSUB Shape
620         RETURN
630 Shape:  ! DRAW IN CURRENT 'WINDOW'
640         PEN -1
650         MOVE 20,20
660         FOR I=0 TO 4
670         IDRAW X(I),Y(I)
680         NEXT I
690         MOVE Ox,Oy
700         PEN 1
710         CALL Cursor(Ox,Oy,P)
720         PIVOT Angle
730         PEN 1
740         FRAME
750         MOVE 20,20
760         FOR I=0 TO 4
770         IDRAW X(I),Y(I)
780         NEXT I
790         RETURN
800 Quit:DISP
810      END
820      !
830      ! ------- SUB PROGRAM ---------
```

```
840      !
850 SUB Cursor(X,Y,P)
860     PEN P
870     PIVOT 0
880     MOVE X,Y
890     IMOVE 5,0
900     IDRAW -10,0
910     IMOVE 5,5
920     IDRAW 0,-10
930     MOVE X,Y
940     SUBEXIT
950 SUBEND
```

SHOWWINDOW



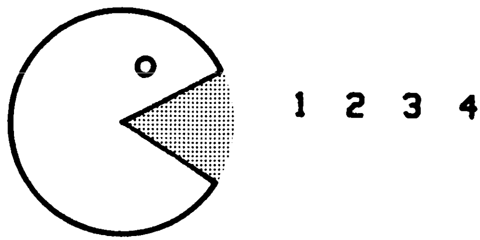

```
10    ! Program "SHOWWINDOW"
20    DIM X(180),Y(180),Prompt$[40],Pad$[40]
30    ! This program compares the mapping of SHOW and WINDOW.
40    ! --- Do all the setup ------------------------------------------------
50    CONTROL CRT,12;1                    ! Turn key labels off
60    Crt_id$=SYSTEM$("CRT ID")
70    Width=VAL(Crt_id$[4,5])
80    Prompt$="New aspect ratio:"
90    Pad$=RPT$(" ",(Width-LEN(Prompt$)) DIV 2)
100   DISP "Calculating the points..."
110   DEG
120   FOR Theta=0 TO 180
130     Radius=COS(5*Theta)  ! (change 5 to another odd number for neat effects)
140     X(Theta)=Radius*COS(Theta)
150     Y(Theta)=Radius*SIN(Theta)
160   NEXT Theta
170   DISP
180   READ Show_left,Show_right,Show_bottom,Show_top
190   READ Model_left,Model_right,Model_bottom,Model_top
200   DATA 0,50,0,30,    57,77,75,95
210   REPEAT
220     GINIT
230     PLOTTER IS CRT,"INTERNAL"
240     GRAPHICS ON
250     Window_right=131
```

```
260   Window_left=Window_right-Show_right
270   Window_bottom=Show_bottom
280   Window_top=Show_top
290   Ratio=(Window_right-Window_left)/(Window_top-Window_bottom)
300   PRINT USING 310;CHR$(12),"Aspect ratio: ",INT(Ratio),FRACT(Ratio)
310   IMAGE K,K,K,.2D
320   ! --- Draw the three plotting surfaces -------------------------------
330   VIEWPORT Model_left,Model_right,Model_bottom,Model_top
340   FRAME
350   VIEWPORT Show_left,Show_right,Show_bottom,Show_top
360   FRAME
370   VIEWPORT Window_left,Window_right,Window_bottom,Window_top
380   FRAME
390   VIEWPORT 0,RATIO*100,0,100
400   ! --- Indicate the Model/Show relationship ---------------------------
410   LINE TYPE 5
420   MOVE Model_left,Model_bottom
430   DRAW Show_left,Show_bottom
440   MOVE Model_left,Model_top
450   DRAW Show_left,Show_top
460   MOVE Model_right,Model_top
470   DRAW Show_right,Show_top
480   MOVE Model_right,Model_bottom
490   DRAW Show_right,Show_bottom
500   ! --- Indicate the Model/Window relationship -------------------------
510   LINE TYPE 6
520   MOVE Model_left,Model_bottom
530   DRAW Window_left,Window_bottom
540   MOVE Model_left,Model_top
550   DRAW Window_left,Window_top
560   MOVE Model_right,Model_top
570   DRAW Window_right,Window_top
580   MOVE Model_right,Model_bottom
590   DRAW Window_right,Window_bottom
600   ! --- Label the various plotting surfaces ----------------------------
610   LINE TYPE 1
620   MOVE Model_left+(Model_right-Model_left)/2,Model_top
630   LORG 4
640   LABEL "Model"
650   MOVE Show_left,Show_top
660   LORG 1
670   LABEL "Show"
680   MOVE Window_right,Window_top
690   LORG 7
700   LABEL "Window"
710   LORG 1
720   ! --- Plot three curves simultaneously -------------------------------
730   FOR Theta=1 TO 180
740     VIEWPORT Model_left,Model_right,Model_bottom,Model_top
750     SHOW -1,1,-1,1
```

```
760       MOVE X(Theta-1),Y(Theta-1)
770       DRAW X(Theta),Y(Theta)
780       VIEWPORT Show_left,Show_right,Show_bottom,Show_top
790       SHOW -1,1,-1,1
800       MOVE X(Theta-1),Y(Theta-1)
810       DRAW X(Theta),Y(Theta)
820       VIEWPORT Window_left,Window_right,Window_bottom,Window_top
830       WINDOW -1,1,-1,1
840       MOVE X(Theta-1),Y(Theta-1)
850       DRAW X(Theta),Y(Theta)
860     NEXT Theta
870     ! --- Ask for the next aspect ratio -----------------------------------
880     DISP Pad$;Prompt$;                   ! Indent the prompt
890     OUTPUT KBD USING "#,K";Pad$          ! Indent the response
900     Ratio=0
910     INPUT "",Ratio
920     IF Ratio>0 THEN
930       IF Ratio>1 THEN
940         Show_right=50
950         Show_top=50/Ratio
960       ELSE
970         Show_top=50
980         Show_right=50*Ratio
990       END IF
1000    END IF
1010  UNTIL Ratio<=0
1020  CONTROL CRT,12;0                       ! Key labels back to default state
1030  PRINT CHR$(12)                         ! Clear the alpha screen
1040  GRAPHICS OFF                           ! Turtn off the graphics screen
1050  END
```

Gload

```
10    ! Program "Gload"
20    OPTION BASE 1
30    INTEGER Return_array(6)
40    GINIT
50    PLOTTER IS CRT,"INTERNAL"
60    SHOW -1,1,-1,1
70    GRAPHICS ON
80    GESCAPE CRT,3;Return_array(*)
90    Size=Return_array(5)*Return_array(6)
100   ALLOCATE INTEGER P0(Size),P1(Size),P2(Size),P3(Size),P4(Size)
110   DEG
120   POLYGON 1,FILL
130   AREA PEN -1
140   MOVE .1,.5
150   PDIR 0
160   POLYGON .1,FILL
170   FOR I=0 TO 4
180     IF I>0 THEN
190       PLOT 0,0
200       PDIR -I*6
210       POLYGON 1,60,I*2,FILL
220     END IF
230     SELECT I
240       CASE 0
250         GSTORE P0(*)
260       CASE 1
270         GSTORE P1(*)
280       CASE 2
290         GSTORE P2(*)
300       CASE 3
310         GSTORE P3(*)
320       CASE 4
330         GSTORE P4(*)
340     END SELECT
350   NEXT I
360   LOOP
370     GLOAD P0(*)
380     GLOAD P1(*)
390     GLOAD P2(*)
400     GLOAD P3(*)
410     GLOAD P4(*)
420     GLOAD P3(*)
430     GLOAD P2(*)
440     GLOAD P1(*)
450   END LOOP
460   END
```


o
1 2 3 4

Index

a

b

C

d

e

f

g

h

i

k

l

m

n

o

p

r

S

t

u

V

W

x

y

MANUAL COMMENT CARD

BASIC 5.0/5.1
Graphics Techniques

HP Part Number 98613-90032 11/87

Please help us improve this manual. Circle the numbers in the following statement that best indicate how useful you found this manual. Then add any further comments in the spaces below. **In appreciation of your time, we will enter your name in a quarterly drawing for an HP calculator.** Thank you.

The information in this manual:

Is poorly organized	1	2	3	4	5	Is well organized
Is hard to find	1	2	3	4	5	Is easy to find
Doesn't cover enough	1	2	3	4	5	Covers everything
Has too many errors	1	2	3	4	5	Is very accurate

fold ——

Particular pages with errors? ______________________________

____

Comments: ______________________________

____

____

____

____

____

____

Name: ______________________________

Job Title: ______________________________

Company: ______________________________

Address: ______________________________

____

☐ Check here if you wish a reply.

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
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

HEWLETT PACKARD

HP Part Number
98613-90032

Microfiche No. 98613-99032
Printed in U.S.A. 11/87

98613-90675
For Internal Use Only