Tektronix®
COMMITTED TO EXCELLENCE

Please Check for
CHANGE INFORMATION
at the Rear of this Manual

PLOT 50

INTRODUCTION TO GRAPHIC PROGRAMMING IN BASIC

INSTRUCTION MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

MANUAL PART NO.
070-2059-01

First Printing DEC 1975
Revised JUL 1981

PRODUCT 4051, 4052, 4054 Graphic Computing Systems

This manual supports the following versions of this product: Version 1 and up

MANUAL REVISION STATUS

REV.	DATE	DESCRIPTION
@	12/75	Original Issue
A,B	9/78	Revised Pages
B	7/79	Revised Pages
B,C	4/80	Revised Pages

WE KNOW YOU'RE ANXIOUS TO LEARN ALL ABOUT THE GRAPHIC SYSTEM, BUT...

you'll miss valuable information if you don't start at the beginning! No matter what your objectives are, you should begin by reading the introduction in the Graphic System Operator's Manual. It presents an overview of the complete Graphic System documentation package, and it will help you select the study material you need to use the Graphic System effectively.

CONTENTS

INTRODUCTION

MATERIAL COVERED

The major part of this manual discusses how to use the Graphic System to make graphs of data and functions. Some additional data, covering graph enhancements, drawn pictures and three-dimensional transformations, is also included. The material covered is appropriate for a person with no experience in graphic programming and only a brief introduction to BASIC.

The programs shown are intended to be easily understood examples. Many of them can be made smaller, faster or otherwise better. Each example program is not presented as the best way to perform a given task, only as a way which is easy to understand.

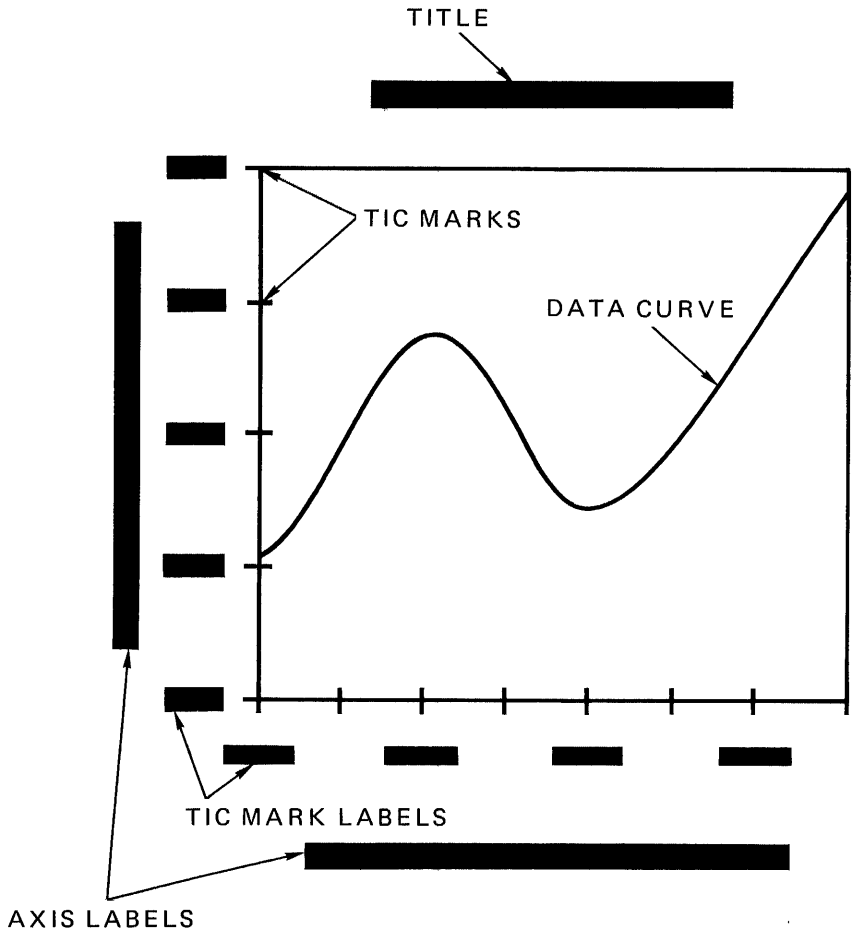
Although this manual can be used as a quick reference guide, it is primarily written to be instructional. Each section assumes that all preceding sections have been read.

GRAPH NOMENCLATURE

Below is a typical graph which includes many features of a complete graph. The data curve itself, although very important, is among the simpler components. Second in importance are the axes and the tic marks which can be added. Major tic marks can be extended to become a grid system; minor tic marks can be added to improve clarity.

The remaining additions all involve characters, either letters or numbers.

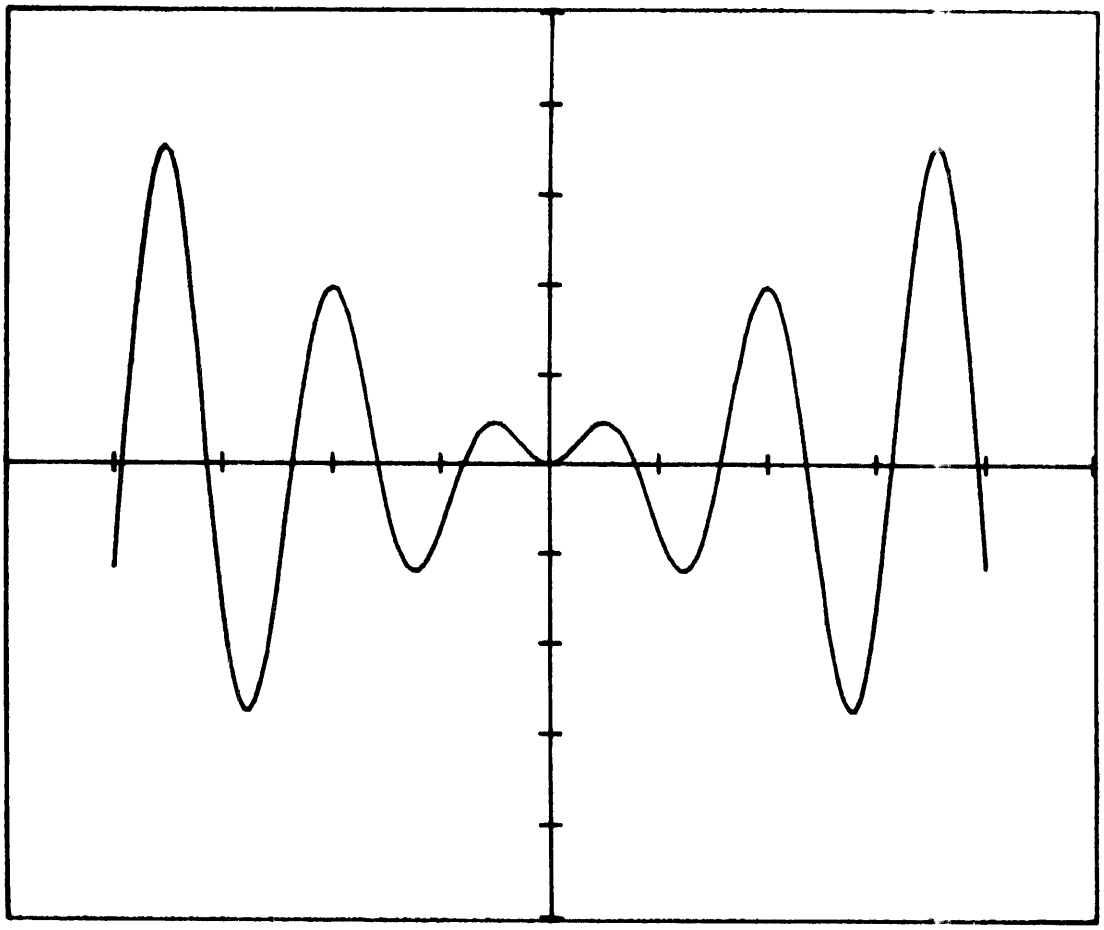
The tic marks can be labeled. Labels on each axis indicate the meaning of the tic labels. The entire graph can be titled as shown.



EXAMPLE

The following is an example to illustrate how Graphic System statements are used to make a simple graph. (The auto line number feature makes entering the program much easier.)

```
100 INIT
110 WINDOW -100,100,-100,100
120 AXIS 20,20
130 MOVE -80,-23
140 FOR I=-80 TO 80
150 DRAW I,SIN(I/5)*I
160 NEXT I
170 END
```

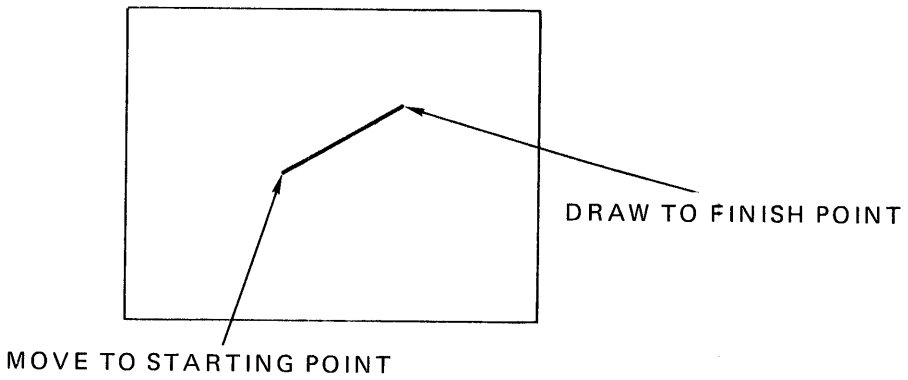


Section 1

GRAPHIC STATEMENTS

MOVE AND DRAW

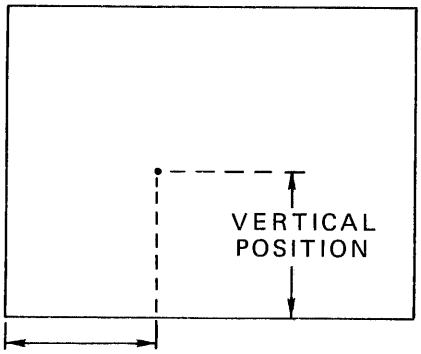
The fundamental building block of drawn information is the line, just as the fundamental building block of written information is the letter. To draw a line on a piece of paper, the pen is first moved to the line's starting point. The pen is placed on the paper and the line is drawn to the endpoint. Drawing a line on the Graphic System (GS) is the same process. The first step is to move to the line's starting point. The second step is to draw to the line's endpoint.



The Graphic System commands for these tasks are MOVE and DRAW.

[Line number] MOVE / DRAW [I/O address] X coordinate in user data units , Y coordinate in user data units

When drawing a line (sometime called a vector) on the GS, the starting and ending points of the line must be specified. A point on the display is located by specifying a horizontal position and a vertical position. As a result, each MOVE and DRAW command has two arguments: one for the horizontal position and one for the vertical position. By convention, the argument specifying the horizontal position is the first of the pair, and the argument specifying the vertical position is the second.



The arguments of the MOVE and DRAW commands can be any of the following:

NUMERIC CONSTANT	Examples:	1.5 -.0009 5025.45 -1.34E207
VARIABLE	Examples:	A B3 Z(68)
EXPRESSION	Examples:	A+1 -R+B*((COS(G)/U)*T)

MOVE and DRAW alter the position of the GRAPHIC POINT. The position of the graphic point is analogous to the position of a pen on a plotter. When no program is running, the graphic point's position is shown by the blinking rectangular cursor.

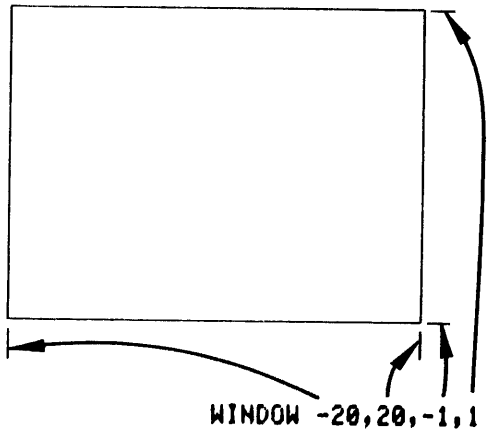
WINDOW

The WINDOW command defines the limits of the data space that the Graphic System display is "looking at". The person using the GS defines what these data limits are. For this reason, the units in the MOVE, DRAW, and WINDOW commands are called USER DATA UNITS in this manual. Similarly, the limits defined in the WINDOW command form what is called a USER DATA SPACE.

Before a graph is drawn on a sheet of paper, the paper must be marked off, both horizontally and vertically, into units appropriate for the information to be graphed. For instance, the same piece of graph paper can be used to show the progress of a glacier moving a few centimeters per year or a jet aircraft moving many kilometers per second. Before a graph is drawn on the display of the GS, the same marking process must be done. The WINDOW command accomplishes this.

[Line number] WINDOW minimum horizontal (X) value in user data units , maximum horizontal (X) value in user data units , minimum vertical (Y) value in user data units , maximum vertical (Y) value in user data units

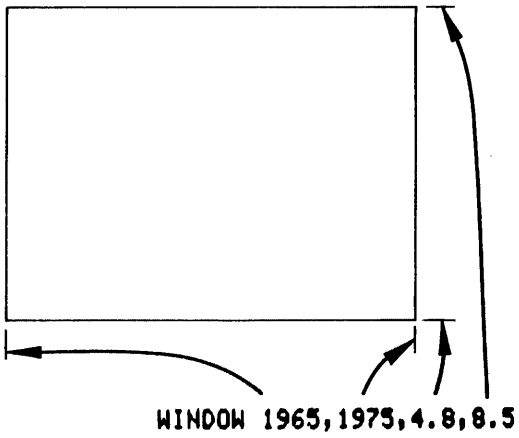
The WINDOW command tells the GS what data values mark the boundaries of the display. For example, the graph program in the Introduction included the following statement:



The WINDOW command tells the GS how to interpret the values specified in the MOVE and DRAW commands. After the above WINDOW command is executed, a MOVE −20,−1 command will place the graphic point at the lower left corner of the display, a MOVE 20,1 command will place the graphic point in the upper right corner of the display, and a MOVE 0,0 command will place the graphic point in the exact middle of the display. WINDOW always requires four arguments, just as MOVE and DRAW always require two arguments.

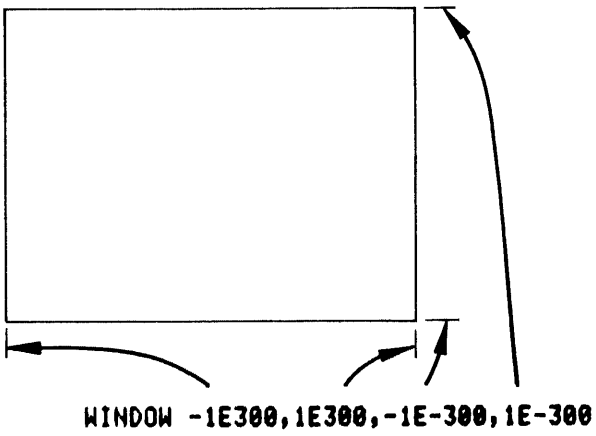
As with MOVE and DRAW commands, the WINDOW arguments can be constants, variables, or expressions. There is no limit to the values these arguments can be, other than the numeric limits of the GS itself.

For example, here is a valid WINDOW command:



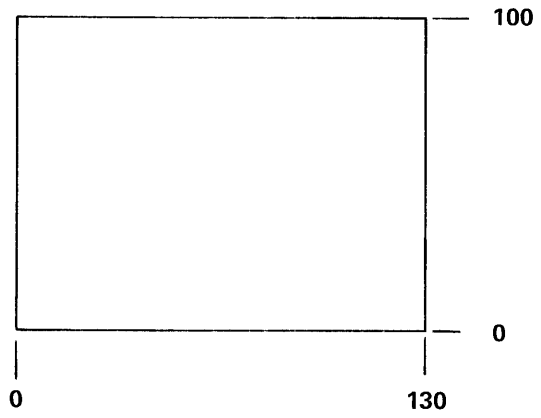
(This WINDOW command might be defining years on the horizontal axis and interest rates on the vertical axis.)

After this command is executed, placing the graphic point at the lower left corner of the display requires a MOVE 1965, 4.8. Here is another valid WINDOW command:



After this command is executed, placing the graphic point at the lower left corner of the display requires a MOVE −1E+300,−1E−300. The horizontal data minimum (the first argument of the WINDOW command) must always be less than the horizontal data maximum (the second argument). The vertical data minimum (the third argument) must always be less than the vertical data maximum (the fourth argument).

The WINDOW command also tells the GS how to interpret the arguments of MOVE and DRAW commands. How does the System interpret these arguments when no WINDOW command has been executed, such as when it is first turned on? To handle this situation, the GS establishes default window limits as shown below.



Whenever it is turned on, the System in effect executes the following command:

```
WINDOW 0,130,0,100
```

(The origin of these numbers is discussed later in this section.) These default window limits are also defined whenever any of the following commands are executed:

```
INIT

OLD

DELETE ALL
```

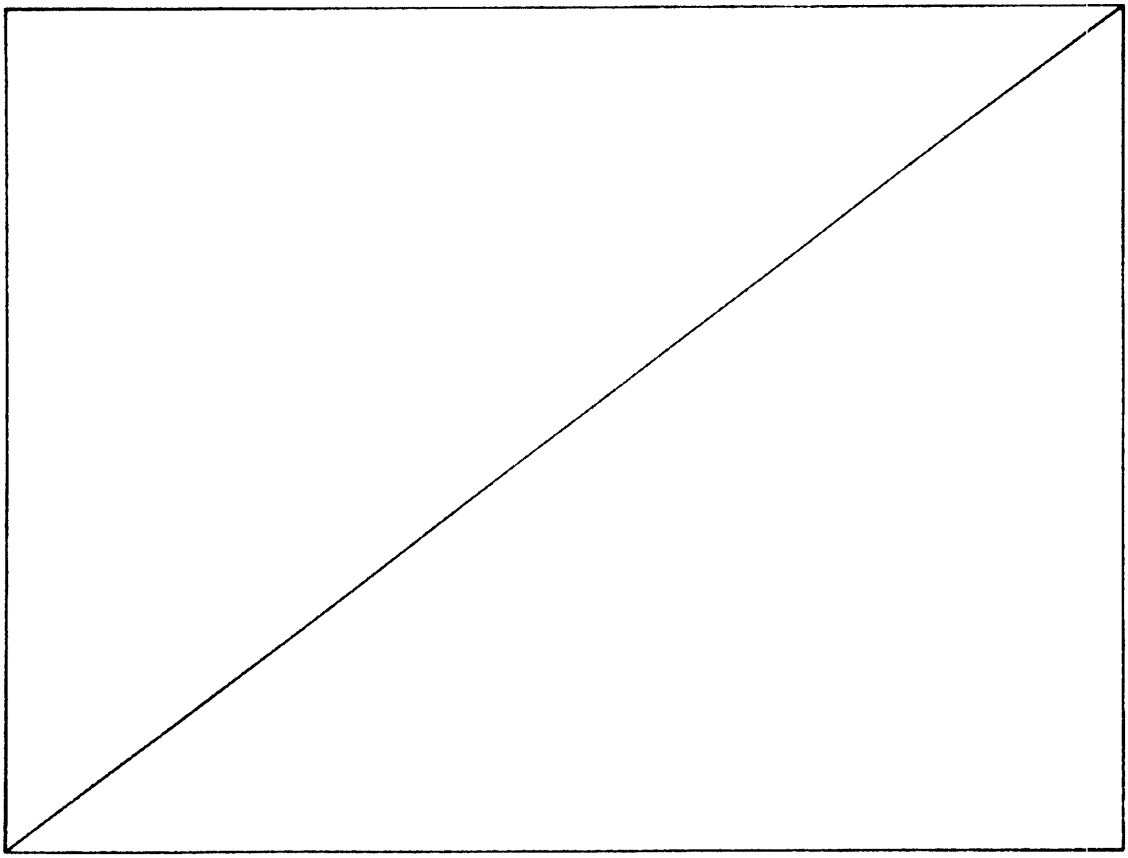
WINDOW limits which differ from these default values are usually desired. A WINDOW command defining the desired limits must be executed after any of the following events occur:

- The GS is turned on
- Any of these commands are executed:

```
INIT
OLD
DELETE ALL
```

The example program shown below will illustrate some of these concepts. After pressing the PAGE key, enter the following statements into the GS:

```
DELETE ALL
100 PAGE
110 INIT
120 MOVE 0,0
130 DRAW 130,100
140 HOME
150 END

RUN
```

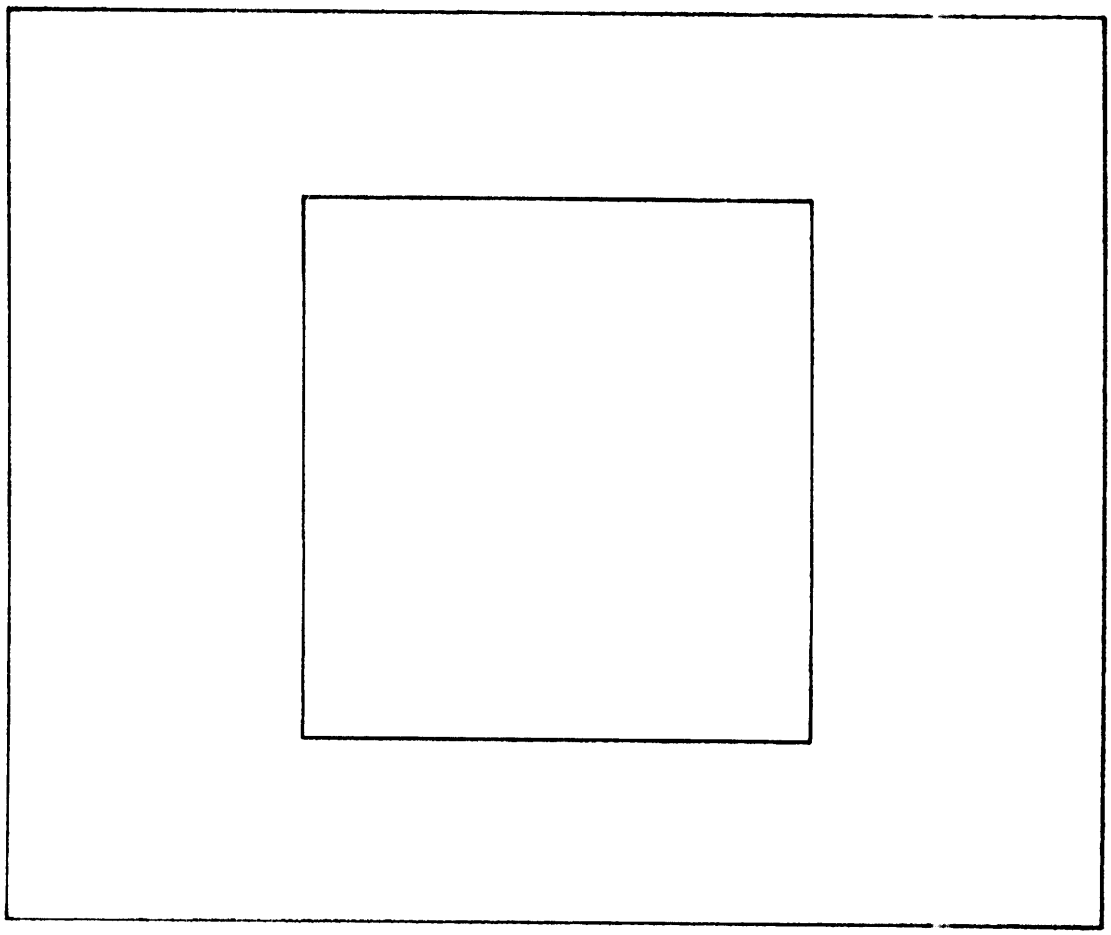


The GS will erase its display and then draw a line from the lower left corner to the upper right corner. The actual positioning of the graphic point during execution of this program can be observed by entering GOTO 100 into the System and then repeatedly pressing the "STEP PROGRAM" key. (This technique is useful for finding errors in complex programs.)

A series of example programs will now demonstrate how WINDOW affects the display. First press the PAGE key (this clears the display and places the cursor in the upper left corner, allowing space for the program to be entered). Now enter the following program:

```
DELETE ALL
INIT
100 PAGE
110 MOVE 35,20
120 DRAW 35,80
130 DRAW 95,80
140 DRAW 95,20
150 DRAW 35,20
160 HOME
170 END

RUN
```

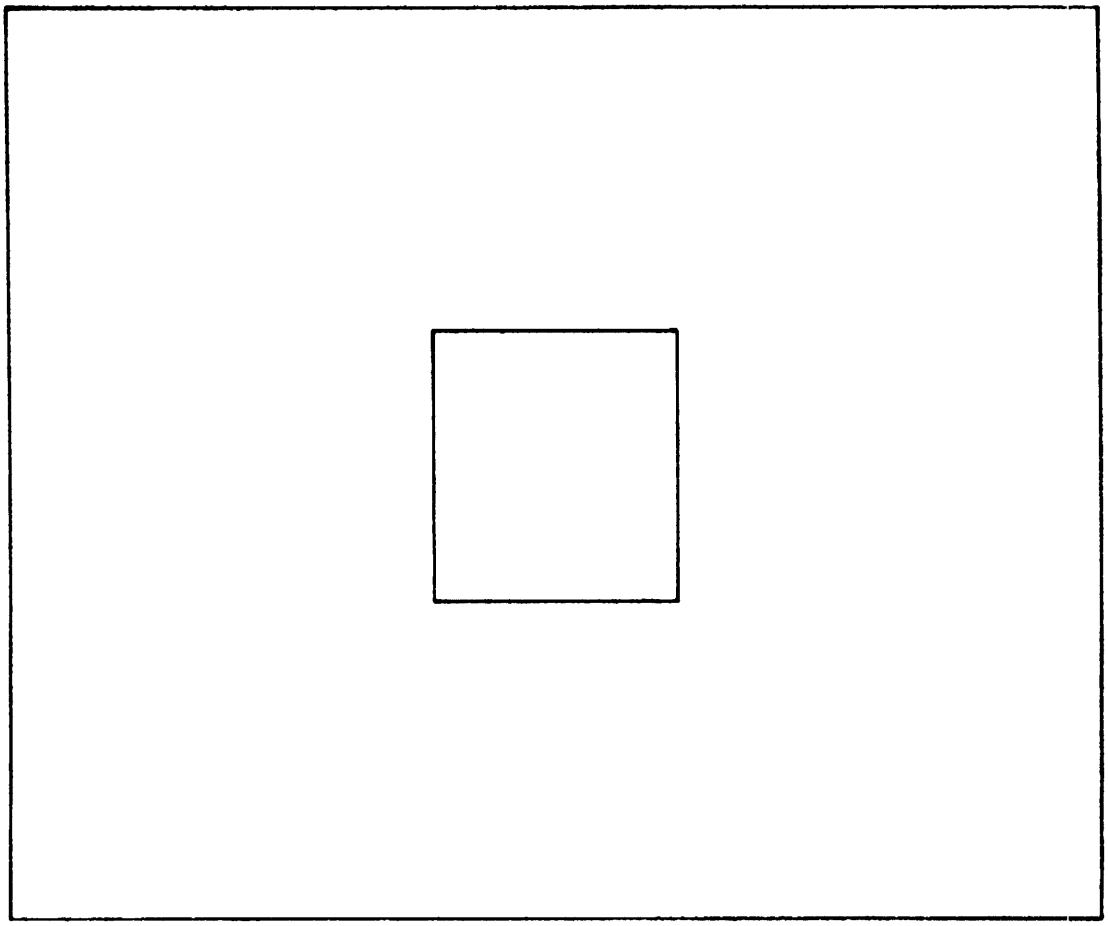
This program draws a square box centered on the screen.



Now enter the following:

```
WINDOW -70,200,-50,150
RUN
```

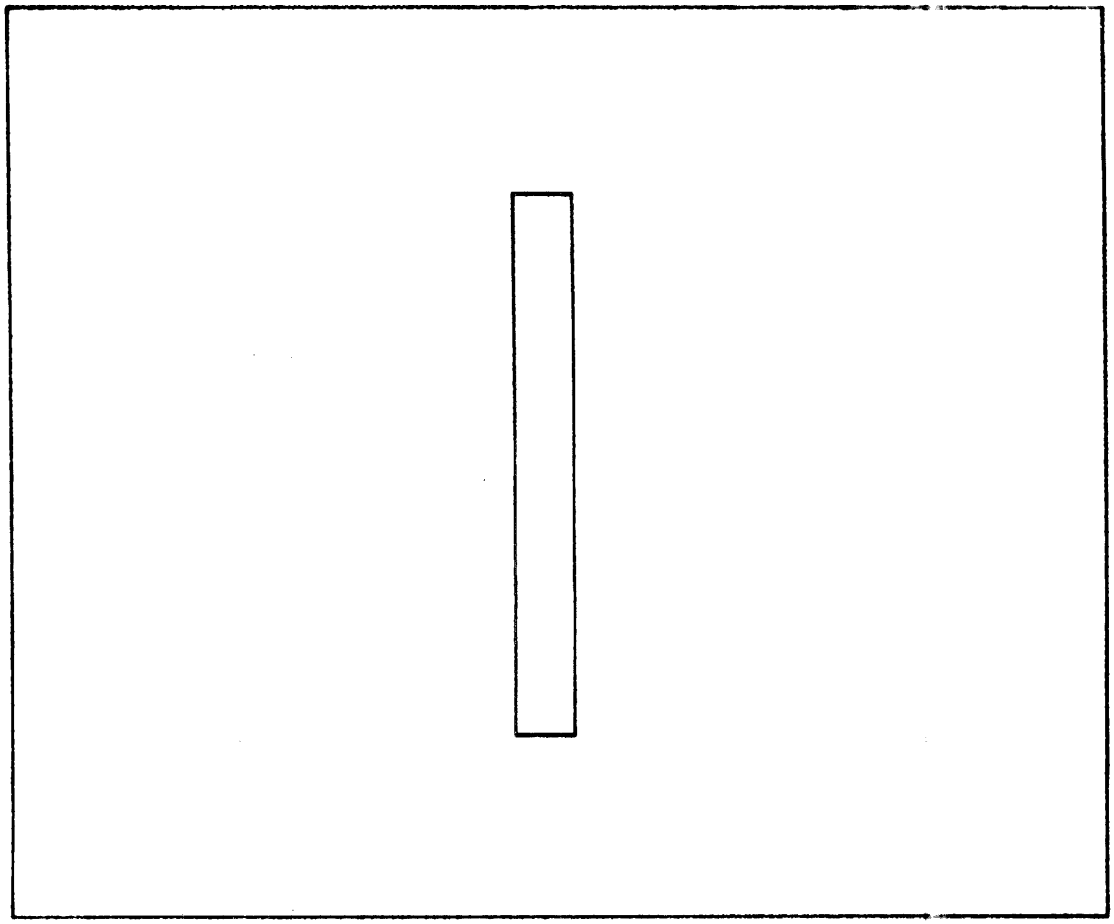
The result is a square box with each edge measuring half the length of the one previously on the screen. It is half size because the WINDOW statement defined the edges of the user data space to be twice as far apart as they were at the default values. The window defined by the above WINDOW statement extends from −70 to 200 user data units on the horizontal axis and from −50 to +150 user data units on the vertical axis. So the screen is, in effect, looking at four times as much area.



With the WINDOW statement, the display can be told to "look at" any rectangular area in user data space. Enter the commands shown below:

```
WINDOW -500,600,0,100
RUN
```

This produces a box the same height as the first box but very narrow. This occurs because the last WINDOW statement defined the data space the screen is "looking at" to be 1100 user data units wide, almost ten times wider than the width implied on the default value. Thus, the box looks very narrow even though its width in data units has not been changed. Note that the height is unchanged from the first time it was displayed because neither the box height in user data units nor the window height in user data units is different in this example. This demonstrates how to change the horizontal and vertical arguments in the WINDOW statement independently. They are totally independent.

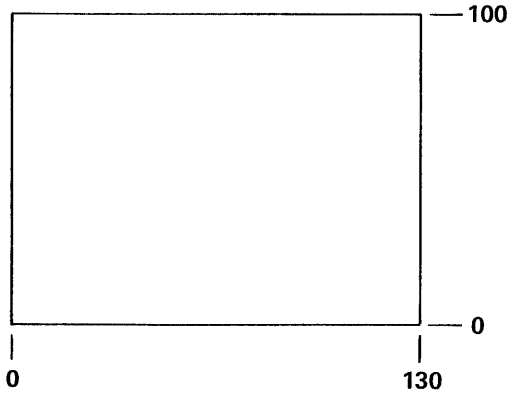


VIEWPORT

When a graph is drawn on a piece of graph paper, it is not always appropriate for the graph to totally fill the paper on which it is drawn. There are times when space is needed around the edges for labeling or titling. In other situations more than one graph is to be drawn on a single piece of paper. These same situations can arise when drawing graphs on the Graphic System. The command used to handle them is VIEWPORT.

[Line number] VIEWPORT minimum horizontal value in GDU's , maximum horizontal value in GDU's , minimum vertical value in GDU's , maximum vertical value in GDU's

The VIEWPORT command is used to specify the size and location of an image on the GS display (and any other graphic device in the 4050 family). In the VIEWPORT command, the display of the GS is considered to be a rectangle 130 units wide by 100 units high.



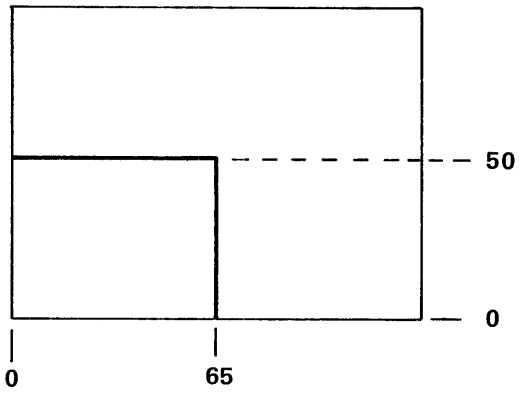
These units are called GRAPHIC DISPLAY UNITS. Unlike user units, Graphic Display Units (GDU's) are fixed for a given device and never change. A horizontal GDU on the display represents the same distance as a vertical GDU on the display. GDU's are used in the VIEWPORT command to specify the size and position of a graph or image on the display of the GS. As with MOVE, DRAW, and WINDOW, the arguments of VIEWPORT may be constants, variables, or expressions. However, the arguments of the VIEWPORT command are interpreted as being GDU's. All four arguments refer to the actual limits for graphic information on the display. The GS will draw no lines outside these limits. The first argument is the location on the display of the left-most limit for graphic information. The GS will draw no lines to the left of this location.

The second argument is the right-most limit. The third argument is the lower limit. The fourth argument is the upper limit. To specify that an image occupy the whole display, a VIEWPORT command with the arguments shown below is executed:

```
VIEWPORT 0,130,0,100
```

To specify that an image is to occupy only the lower left corner of the display, a VIEWPORT command with the arguments shown below is executed:

```
VIEWPORT 0,65,0,50
```



Just as there are default values for the WINDOW command, there are default values for the VIEWPORT command. The default size for an image is the full display. Whenever the GS is turned on, it in effect executes the following command:

```
VIEWPORT 0,130,0,100
```

This default viewport size is also defined whenever any of the following commands are executed:

```
INIT
OLD
DELETE ALL
```

For this reason, all examples so far in this section have referenced the whole display. They have been run as if a VIEWPORT 0,130,0,100 command had been previously executed. (The default WINDOW limits are 0,130,0,100. This defines a one-to-one correspondence between GDU's and user data units when the GS is first turned on.)

If a viewport is desired which is different from the default size, a VIEWPORT command defining the desired size must be executed after any of the following events occur:

- The GS is turned on
- Any of these commands are executed:

```
INIT
OLD
DELETE ALL
```

For the next two examples, the square box program listed below should be in the GS memory. (Pressing the PAGE key and entering a LIST command will confirm if it is.)

```
100 PAGE
110 MOVE 35,20
120 DRAW 35,80
130 DRAW 95,80
140 DRAW 95,20
150 DRAW 35,20
160 HOME
170 END
```

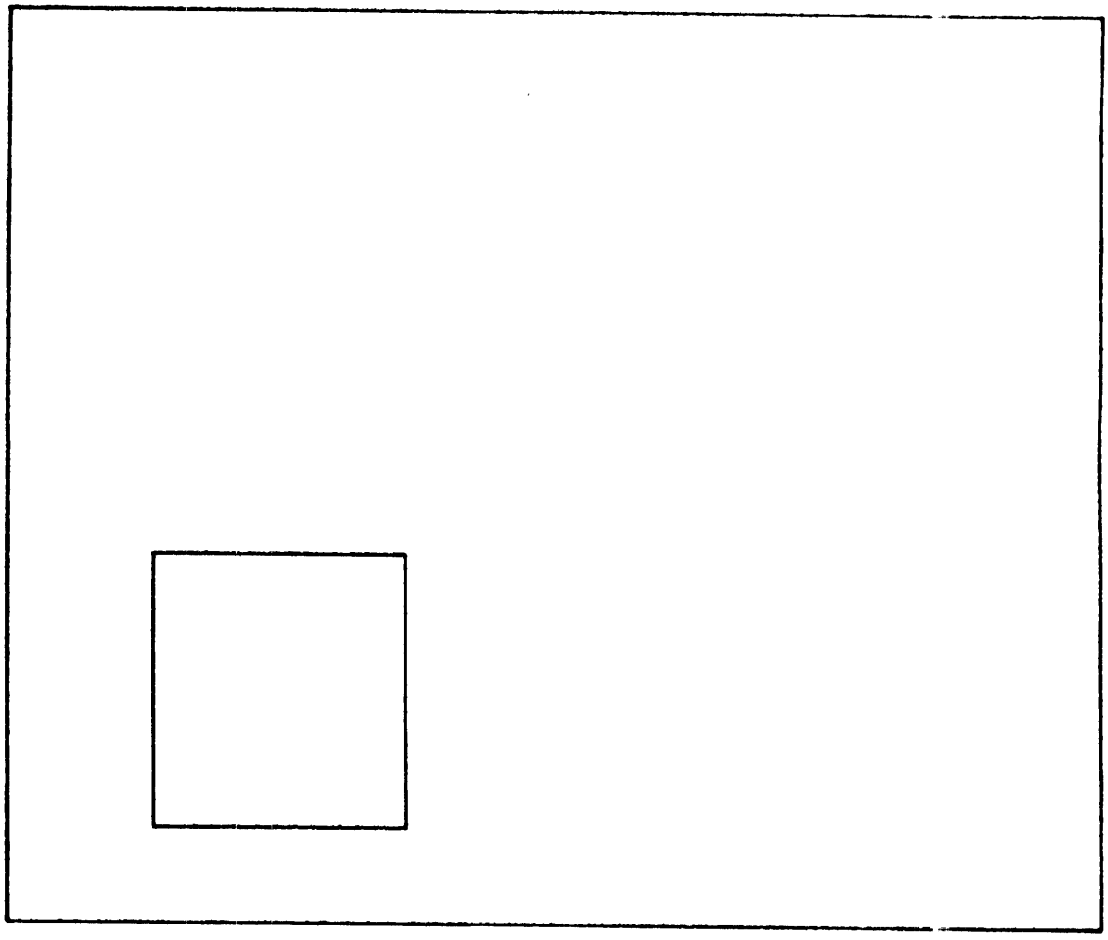
After pressing the PAGE key, enter the following:

```
INIT
RUN
```

The square box should now appear centered on the display. Now enter the following commands:

```
WINDOW 0,130,0,100
VIEWPORT 0,65,0,50
RUN
```

The square box reappears but with a different size and location. The graphic output of this program can be examined two ways. One way is to see what the viewport command has done (shown in the following diagram):

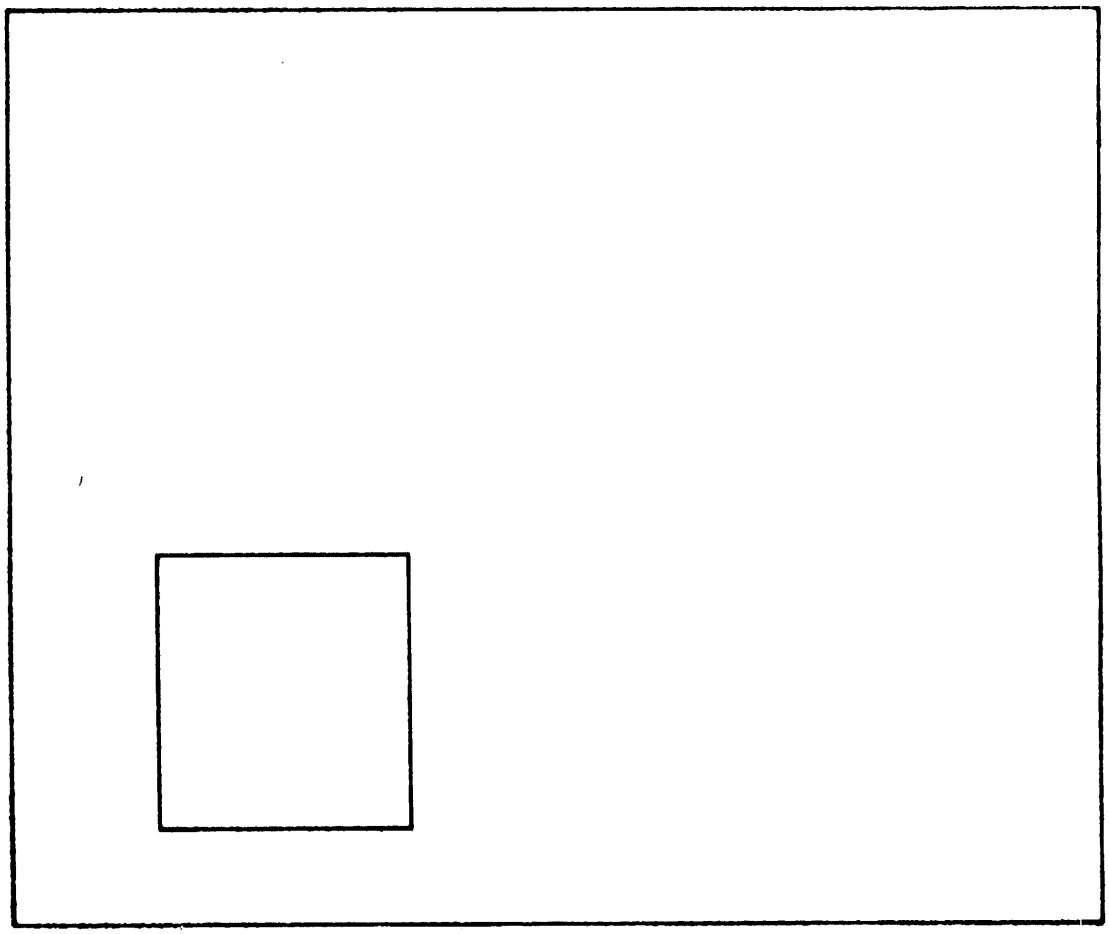


```
VIEWPORT 0,65,0,50
```

What formerly occupied the whole display now occupies only the lower left corner, as specified by the VIEWPORT 0,65,0,50 command, above.

The left edge of the viewport is still the left edge of the display because the first argument is zero (as in the default VIEWPORT 0,130,0,100). However, the right edge of the viewport is now only half way across the display because the second argument is 65 GDU's, half the display width of 130 GDU's. In specifying the vertical size of the viewport, a similar situation exists. The bottom of the viewport is still the bottom of the display but the top of the viewport is defined to be half way up the display (because the last argument in the above VIEWPORT command is 50 GDU's).

Another way to look at the output of this program is to see how the window has been affected. This is shown below:



```
WINDOW 0,130,0,100
```

The same window as in previous examples is now drawn with a different size and location on the display. It is otherwise unchanged.

As with other GS commands, WINDOW and VIEWPORT are executed immediately if entered without a preceding statement number. An entered command becomes part of a stored program if it is entered with a preceding statement number. For example,

```
WINDOW 0,130,0,100
```

is executed immediately while

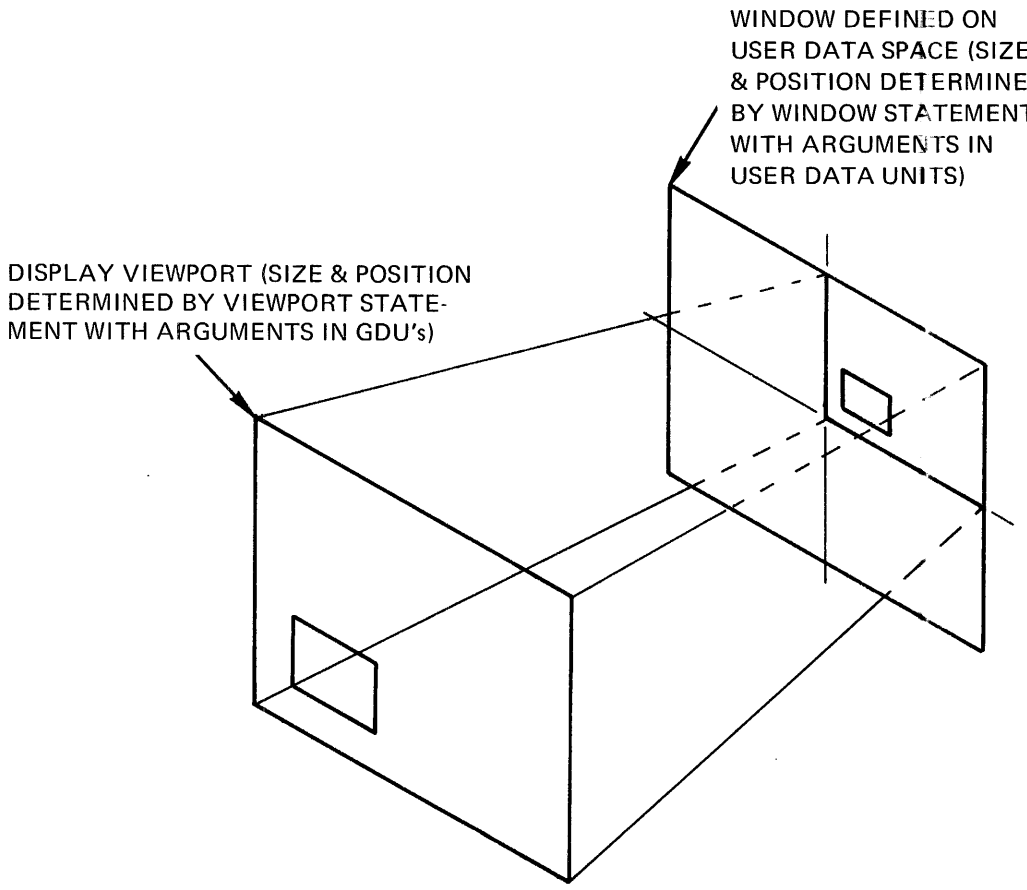
```
250 WINDOW 0,130,0,100
```

becomes statement 250 in whatever program is currently in GS memory.

WINDOW AND VIEWPORT

WINDOW and VIEWPORT accomplish conceptually similar tasks. As shown in the diagram below, the combination of these two statements allow the selection of what is to be seen (WINDOW) and where on the display it is to be located (VIEWPORT). The WINDOW command manipulates the size and location of the window on the user's data space. Its arguments are interpreted as user data units. The VIEWPORT command manipulates the actual size and location of the image on the display. Its arguments are interpreted as Graphic Display Units. WINDOW can be used to "zoom" in on a particular area of a graph to display more detail. VIEWPORT can be used to make several graphs appear on the display at one time.

To repeat: VIEWPORT specifies the physical area on the display to be used for graphic information; WINDOW specifies how many units of measure (that is, user data units) will be included within this physical graphing area.



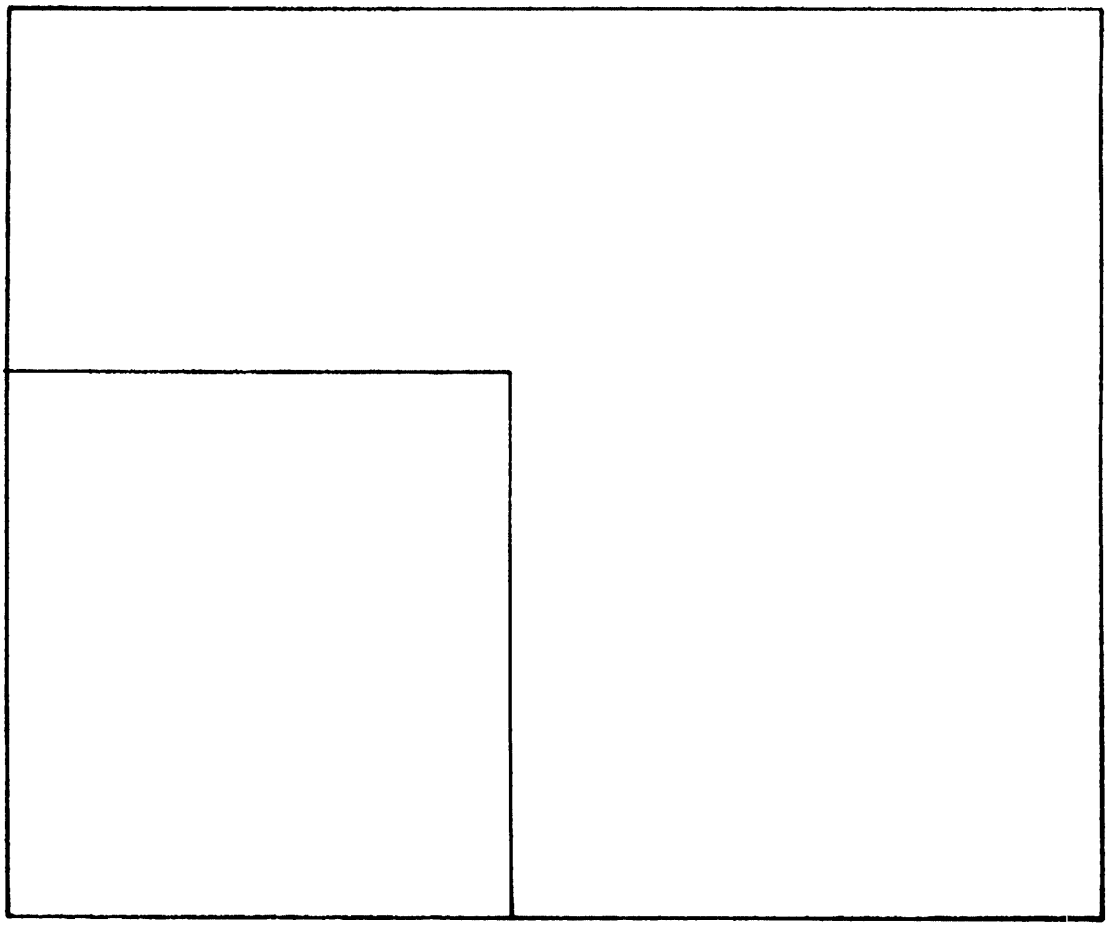
Clipping

These commands (WINDOW and VIEWPORT) also serve another function. As an example, try the following. Confirm that the square box program on page 1-7 is in the GS by entering the following statements:

```
INIT
RUN
```

The square box should again appear on the center of the display. Now enter the following:

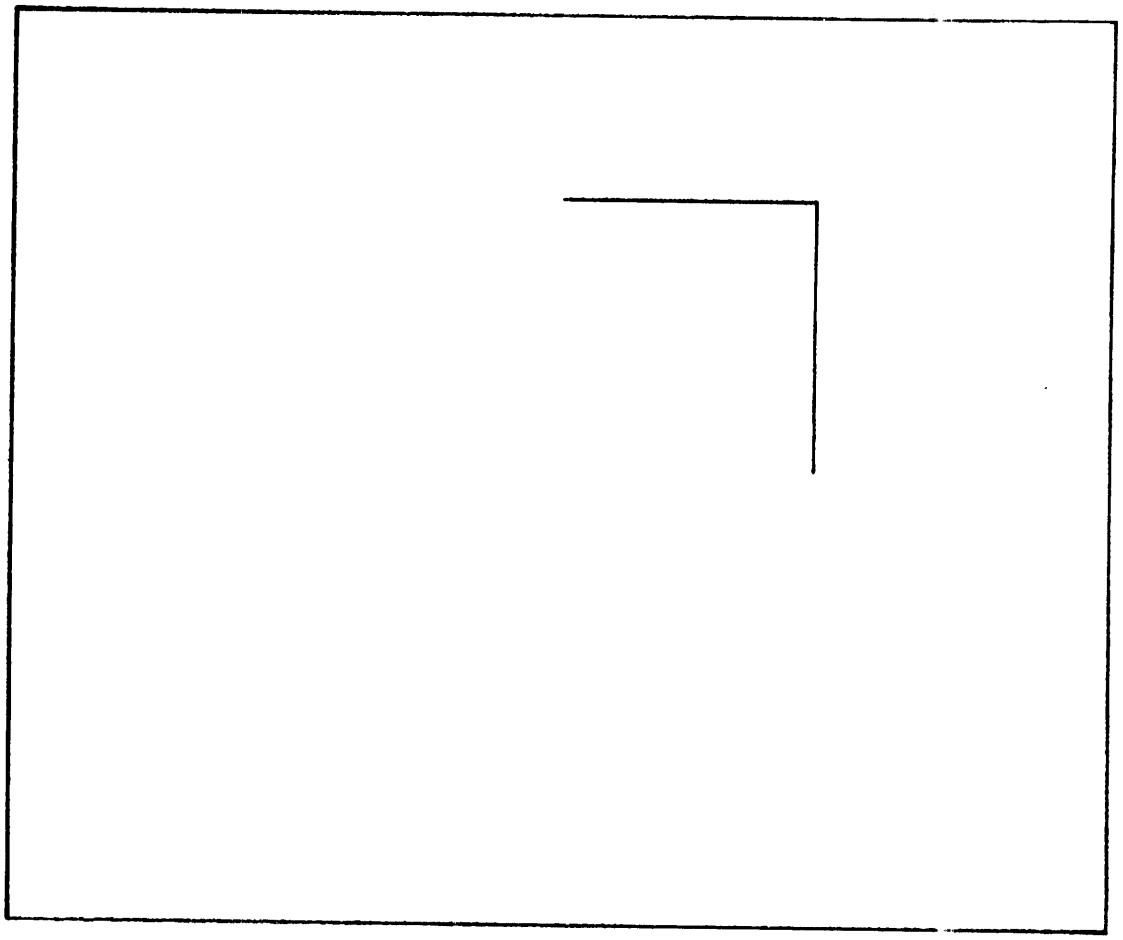
```
INIT
WINDOW 65,130,50,100
RUN
```



Notice now that because of the changed arguments in the WINDOW statement, only the upper right corner of the square appears (although it is filling the display). Because the VIEWPORT statement's arguments are at their default values (0,130,0,100), this upper right corner fills the display.

WINDOW is used to specify what is to be seen and what is not to be seen. Any lines which a program attempts to draw outside the data limits specified in the WINDOW command are not drawn. Now enter the following:

```
VIEWPORT 65,130,50,100
RUN
```



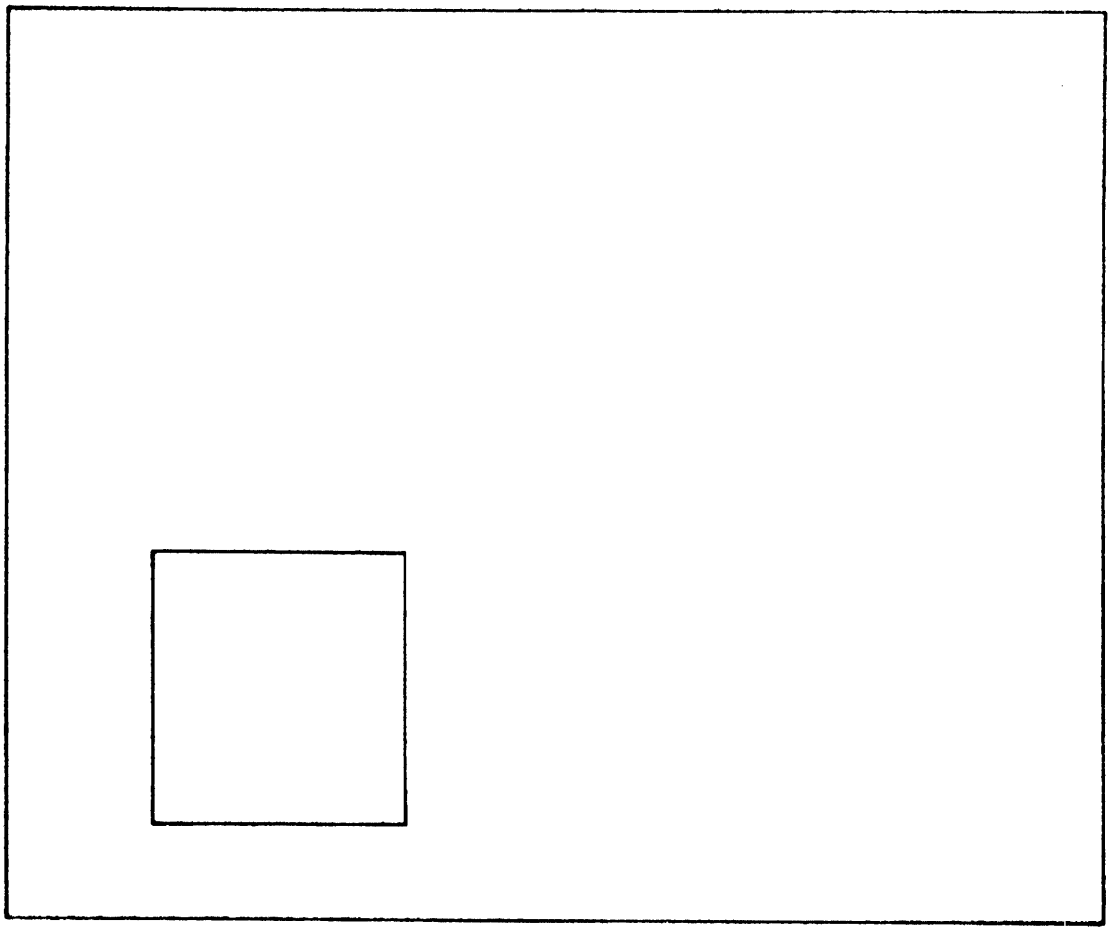
Preventing any drawing outside a specified WINDOW is called clipping. Only the upper right corner of the square appears, located in the upper right quarter of the display. The remainder of the square is not drawn because of the clipping capability of the GS.

SCALE

[Line number] SCALE horizontal scale factor , vertical scale factor

As shown earlier, the WINDOW statement allows the user to define what portion of the data space is to be seen. The SCALE command is an alternative way to do this. In effect, SCALE allows the user to do the same thing that WINDOW does but in a slightly different way. After confirming that the box program on page 1-7 is in memory, enter the following:

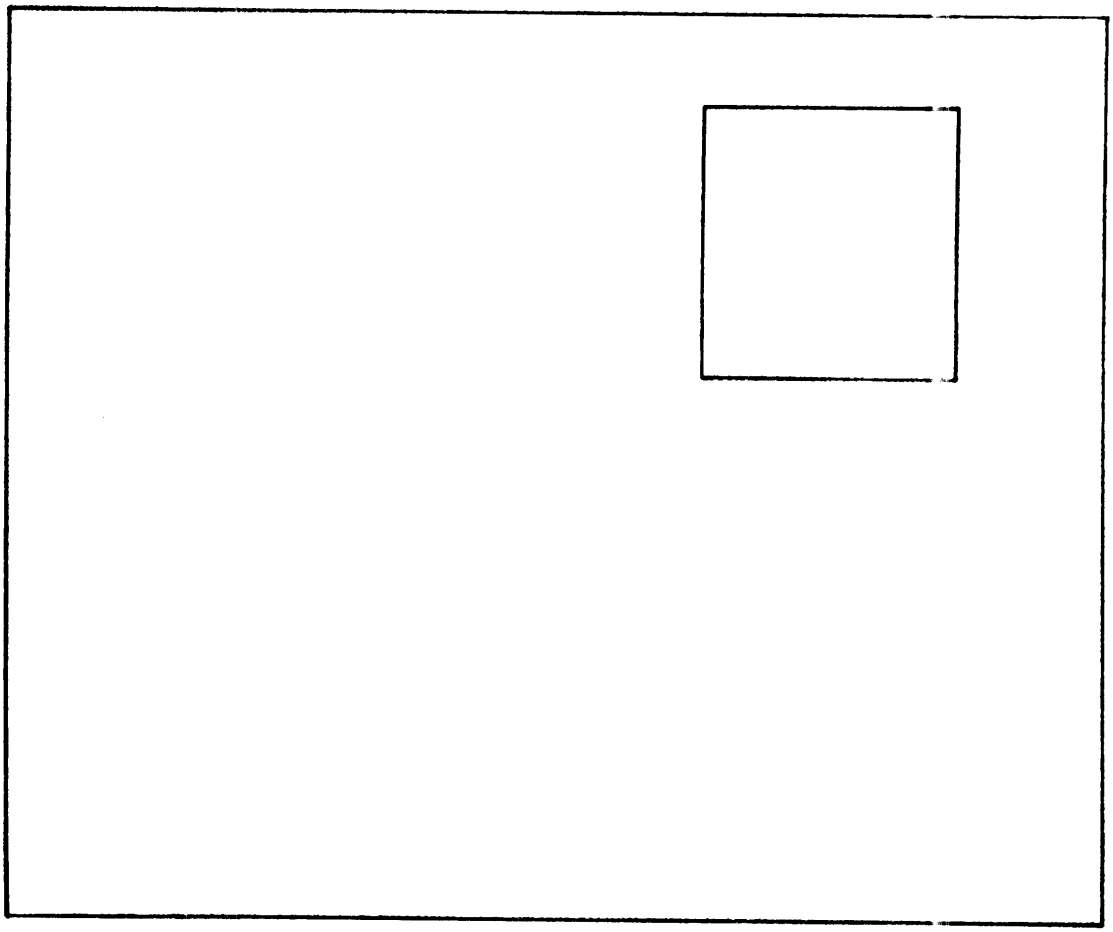
```
INIT
80 MOVE 0,0
90 SCALE 2,2
RUN
```



The box appears half size in the lower left corner of the screen. Each argument in SCALE is interpreted to be the number of user data units which are to be represented by each GDU. The default values for these areguments are SCALE 1,1. SCALE 2,2 indicates that the data should appear half the size it would have appeared with the default argument values. This also implies that, if the VIEWPORT arguments are unchanged, the same viewport on the display should contain four times as much user data space area.

Enter the following:

```
INIT
80 MOVE 65,50
90 SCALE 2,2
RUN
```



The square box is drawn in the upper right quadrant of the screen. The only difference in these two examples is the MOVE command which preceded the SCALE command. These examples demonstrate another function of the SCALE command. SCALE sets the location of the point X = 0 and Y = 0 in user data space to be the current location of the graphic point. This sequence of statements:

```
INIT
MOVE 0,0
SCALE 2,2
```

is equivalent to: WINDOW 0,260,0,200. Similarly:

```
INIT
MOVE 65,50
SCALE 2,2
```

is equivalent to: WINDOW −130,130,−100,100. (The INIT commands in the two examples are not strictly required but are used to ensure that the first MOVE 0,0 is actually a move to the lower left corner of the display and that the second MOVE 65,50 is actually a move to the center of the display.)

The INIT statement is used in this section to ensure that the examples have been run from a common starting point. INIT, in addition to other functions, restores the arguments of the WINDOW, SCALE, and VIEWPORT statements to their default values. These are:

```
WINDOW 0,130,0,100
SCALE 1,1
VIEWPORT 0,130,0,100
```

Because of the INIT command the arguments of the MOVE commands (which immediately preceded the SCALE commands in the examples) were all interpreted the same way.

GRAPHIC OUTPUT

Earlier in this section, the MOVE and DRAW statements were discussed. These statements and the other graphic output statements are now covered in somewhat more detail. All these statements affect the location of the cursor (also called the graphic point). The location of this point is always maintained internally by the GS and, if a program is not running, the GS constantly shows the physical location of this point on the screen. The graphic point is actually located in the lower left corner of the cursor's flashing rectangular dot matrix. This is the starting point for any graphic output operation.

PAGE and HOME

Two functions useful for cursor housekeeping tasks are PAGE and HOME. These commands can be executed either from the keyboard (using the PAGE and HOME key) or under program control (by entering a line number before the words PAGE and HOME). The box program on page 1-7 contains HOME used under program control.

The PAGE command erases the display and moves the cursor to the "home" position. On the display, this is the position of the upper left most character the display is capable of printing. The HOME command, in contrast, only moves the cursor to the home position and does not erase the display. The HOME position is fixed. It is not affected by WINDOW, VIEWPORT, or SCALE.

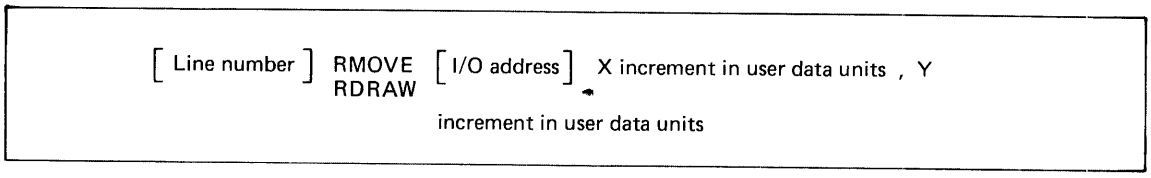
MOVE and DRAW

Execute an INIT command and press the PAGE key. The cursor will appear in the display's upper left corner. A DRAW 0,0 will cause a line to be drawn from the current cursor location to the location X = 0 and Y = 0 in user data units, which, because of the INIT command, is the lower left corner of the screen. The arguments of the DRAW command specify the end point of the line. The execution of a DRAW will cause a line to be drawn from wherever the graphic point is to the location specified in the DRAW command. The resulting line is, of course, subject to the limits imposed by WINDOW. The cursor is moved from the HOME position in the upper left corner of the display to the position of the end point of the line. Press the PAGE key and enter DRAW 64.5,49.5.

This will cause a line to be drawn to a location which is approximately in the middle of the display. The line ends at the cursor's lower left corner. The arguments of the DRAW command (or any other graphic command) need not be integers.

RMOVE and RDRAW

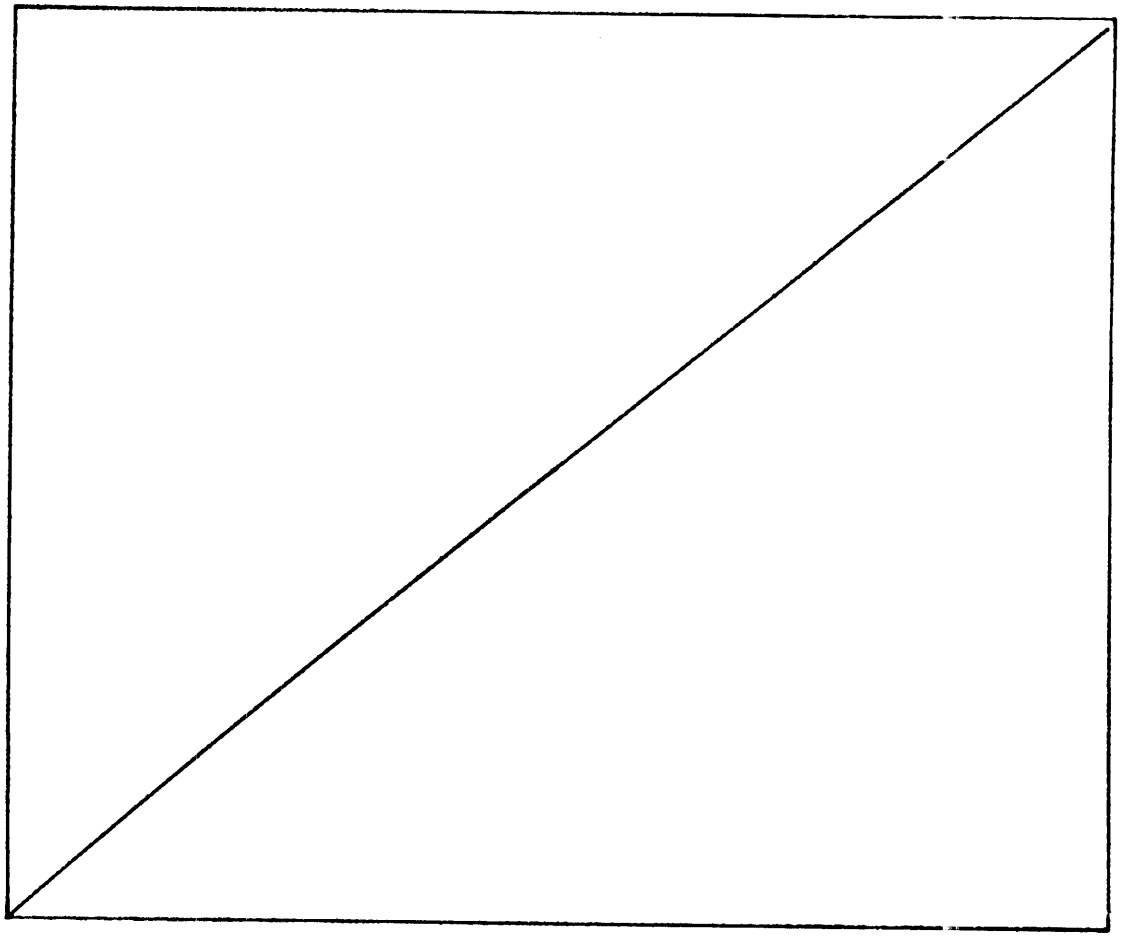
There are two types of MOVE and DRAW commands. The type that has been used so far in this section is called an "absolute" MOVE or DRAW command. These commands cause the cursor or graphic point to be moved from its present location (wherever that happens to be) to the location specified by the command's arguments (in user data units).



The other type of MOVE and DRAW command is called "relative". RMOVE and RDRAW are the relative move and draw commands. Either of these commands will cause the graphic point to be moved relative to its present location. That is, the arguments of the RMOVE and RDRAW commands are interpreted to be the distances (in user data units) to move away from the graphic point's present location. If the command RDRAW 64.5,49.5 is executed, a line is drawn from the current graphic point to a point 64.5 horizontal and 49.5 vertical user data units away from the beginning point of the line.

Enter the following into the machine:

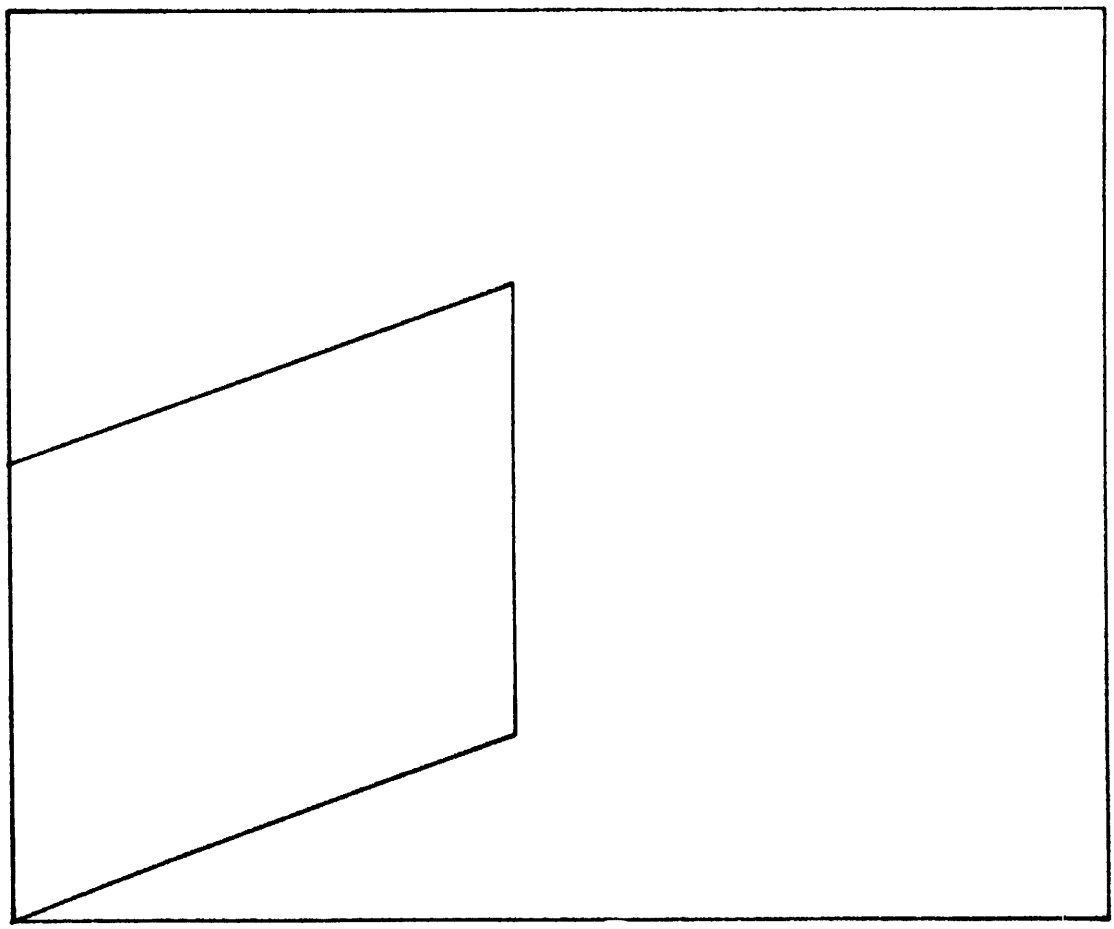
```
DELETE ALL
PAGE
100 INIT
110 MOVE 0,0
120 RDRAW 64.5,49.5
130 RDRAW 64.5,49.5
140 HOME
150 END

RUN
```



Each of the RDRAW commands causes a line to be drawn from the current location of the graphic point to a point 64.5 horizontal and 49.5 vertical data units away.

MOVE and DRAW cause the graphic point or cursor to be moved relative to the origin or 0,0 point of the user data space. The arguments of these commands are treated as if they described an absolute location in user data space. RMOVE and RDRAW cause the graphic point or cursor to be moved relative to its current location. The arguments of these commands are treated as incremental distances from the current point. Another example illustrates these differences. Enter the following into the Graphic System:

```
DELETE ALL
PAGE
100 INIT
110 PAGE
120 MOVE 0,0
130 DRAW 60,20
140 MOVE 0,50
150 RDRAW 60,20
160 DRAW 60,20
170 HOME
180 END

RUN
```



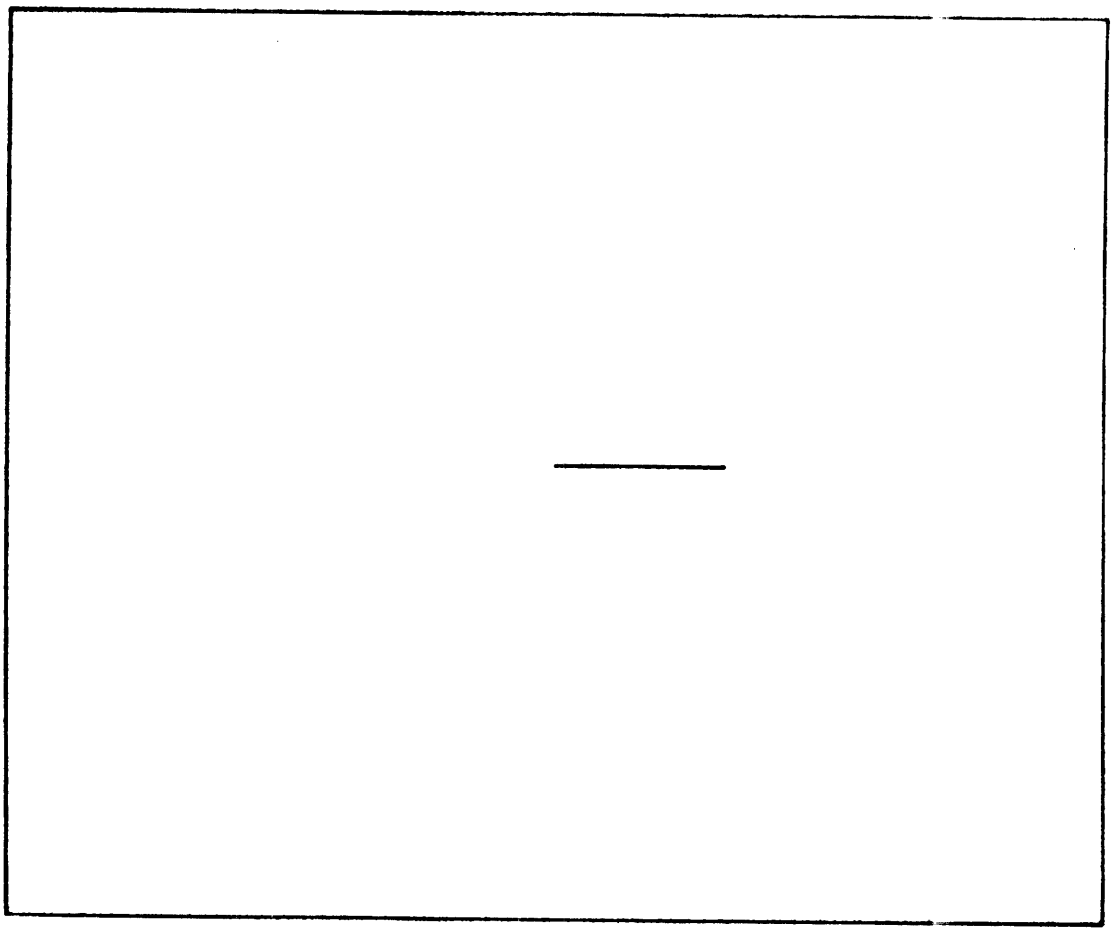
Notice how the command which precedes each of the three DRAW commands affects execution. The first DRAW could have been a RDRAW without affecting the shape of the drawn pattern. This is because its beginning point is also the origin of the user data space coordinate system, that is, the point where X = 0 and Y = 0.

ROTATE

[Line number] ROTATE rotation angle measured in the current trigonometric units

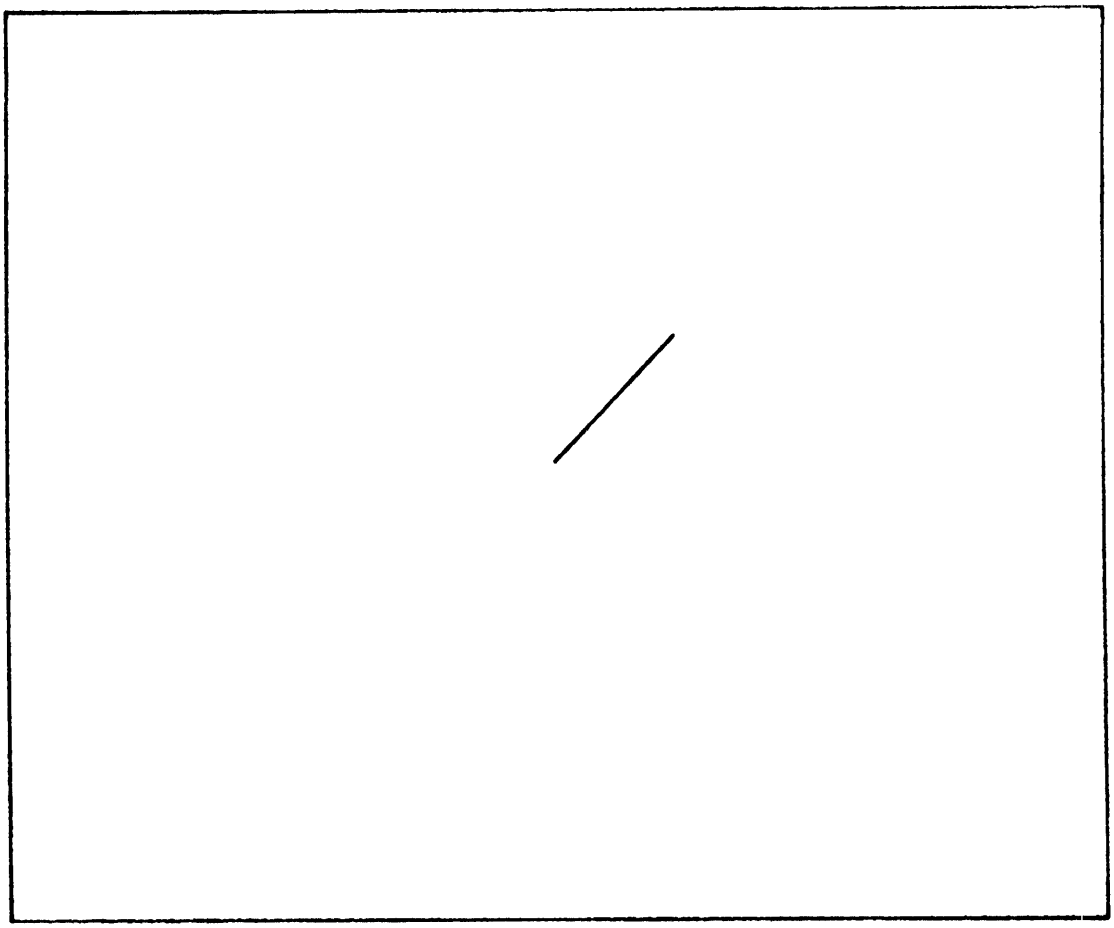
A command which applies only to RMOVE and RDRAW is ROTATE. This command causes each RMOVE and RDRAW to be drawn rotated a specified angle from its expected direction. The point around which each RMOVE or RDRAW is rotated is its starting point. For example; enter the following commands:

```
PAGE
DELETE ALL
INIT
100 PAGE
110 MOVE 65,50
120 RDRAW 20,0
130 HOME
140 END

RUN
```



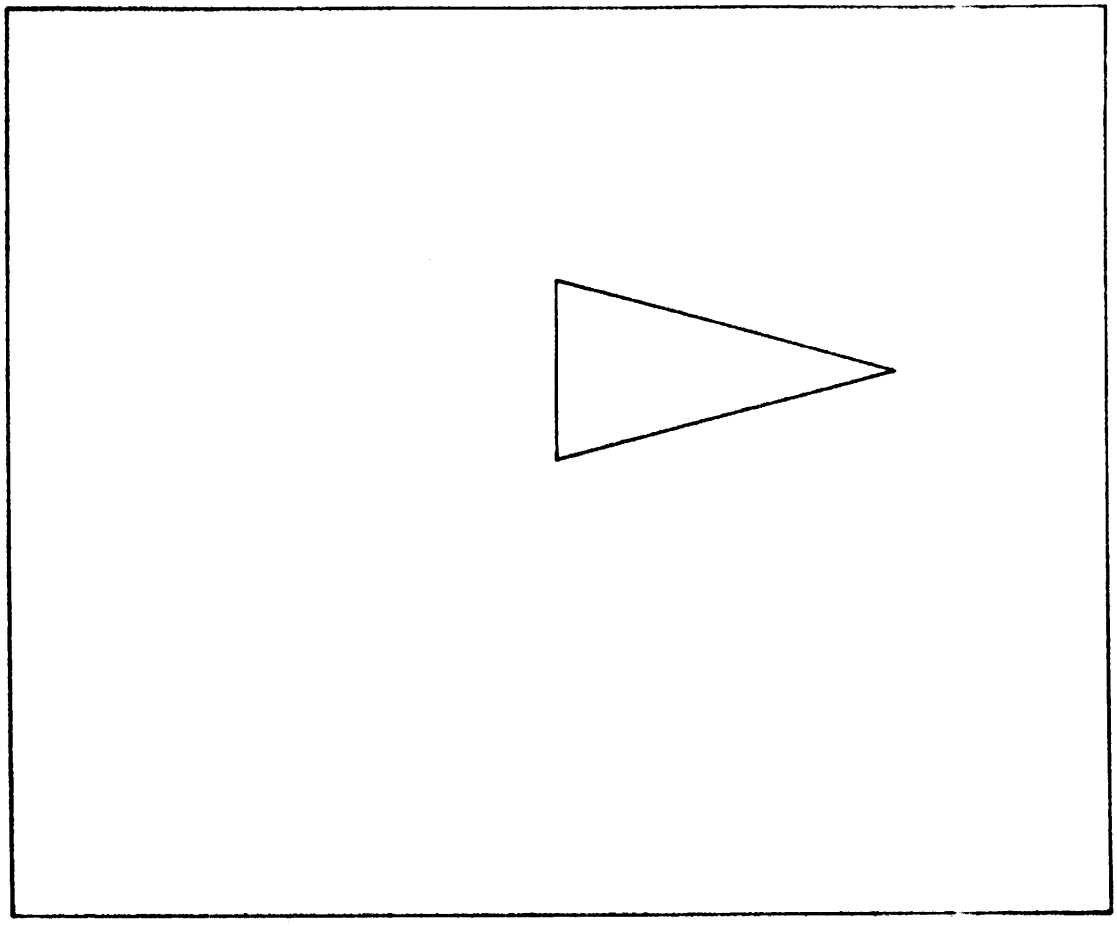
The above program draws a horizontal line 20 data units long with its beginning point centered on the display. Now enter the following:

```
SET DEGREES
ROTATE 45
RUN
```



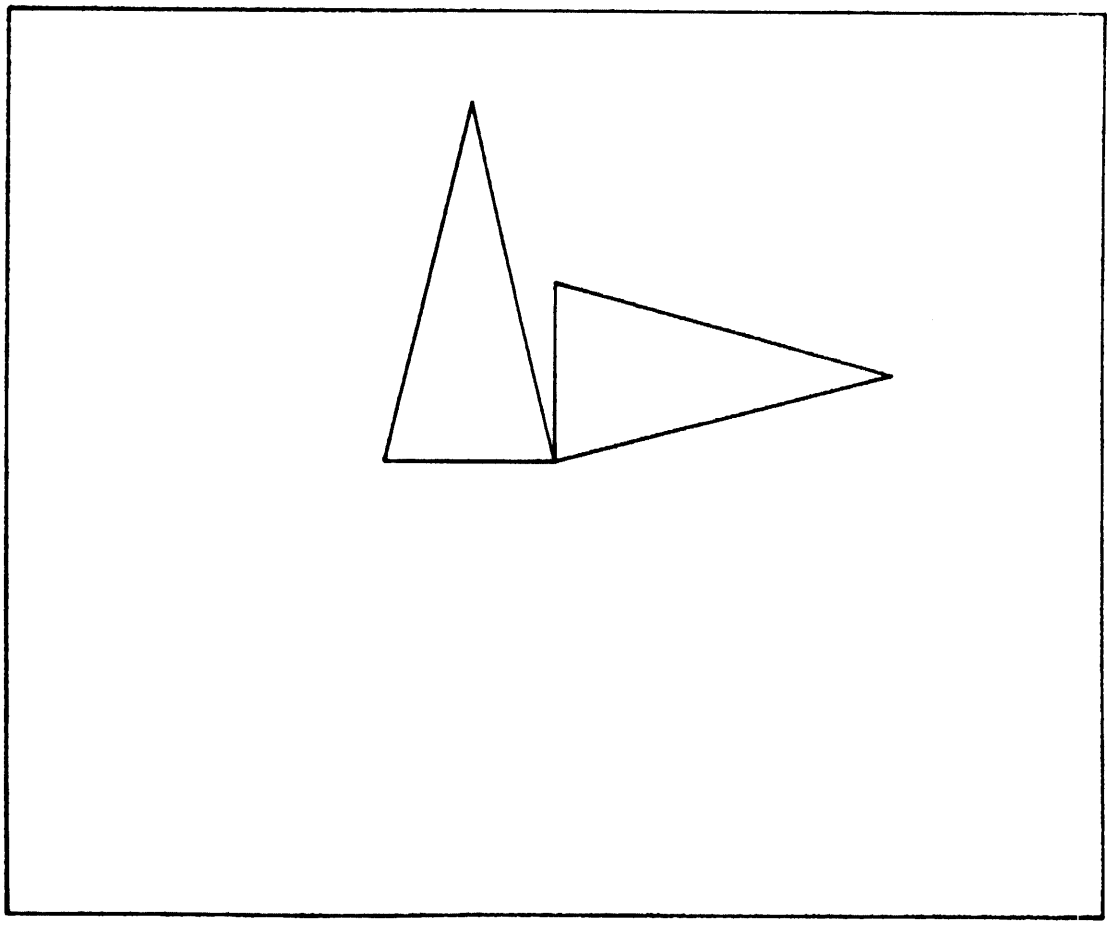
The same RDRAW command is executed, but the resulting line is rotated 45 degrees counter-clockwise around its beginning point. The significant feature about this, other than the rotation, is that the line is rotated around its beginning point. This also happens to be the ending point of an absolute MOVE, which was not rotated. The following example demonstrates the usefulness of this characteristic:

```
DELETE ALL
100 INIT
110 SET DEGREES
120 PAGE
130 ROTATE 0
140 MOVE 65,50
150 REM   DRAW A TRIANGLE
160 RDRAW 40,10
170 RDRAW -40,10
180 RDRAW 0,-20
190 HOME
200 END

RUN
```



The display should now show a narrow isosceles triangle "pointed" to the right. Add the following statements:

```
132 GOSUB 140
134 ROTATE 90
136 GOSUB 140
138 END
200 RETURN

RUN
```

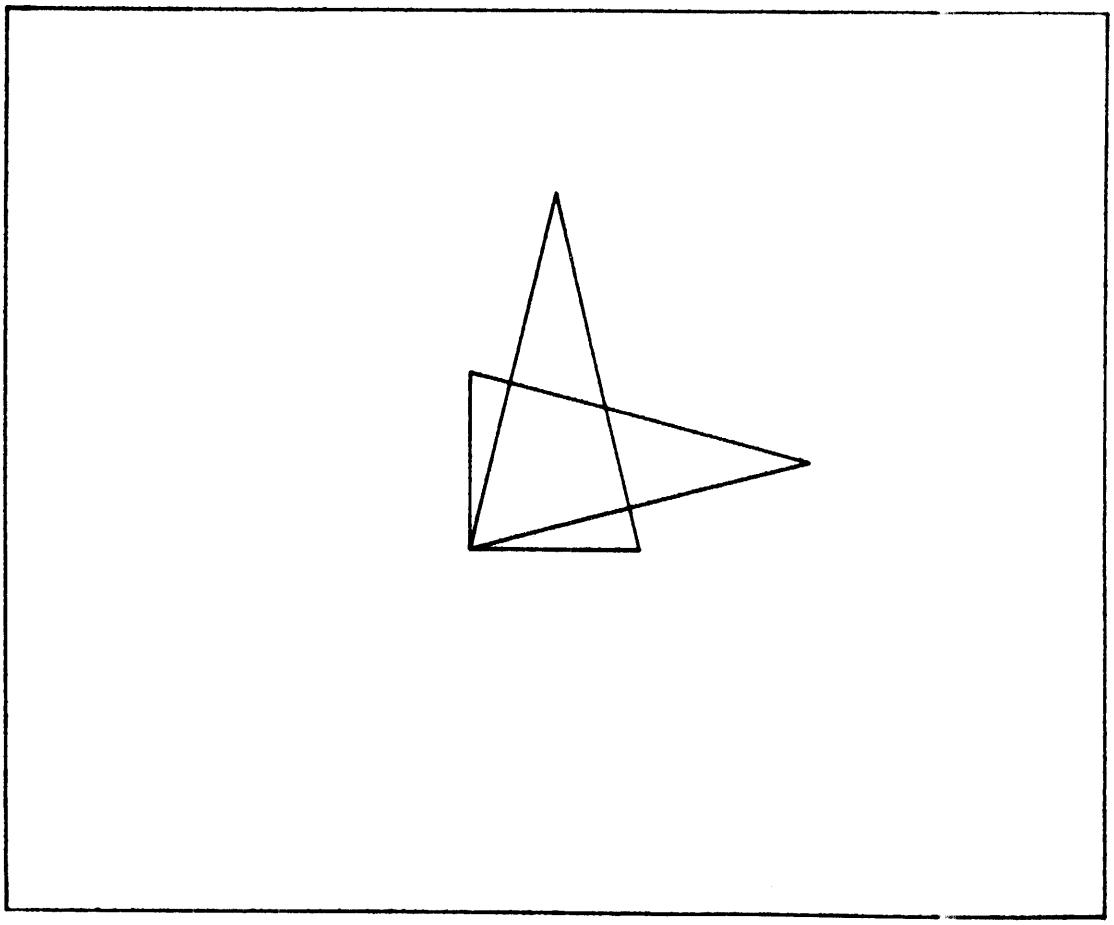


The triangle has been drawn again but rotated 90 degrees counterclockwise around the vertex that corresponds to the endpoint of the MOVE command. This point of rotation can be placed anywhere desired. For example, enter the following statement into the GS:

```
155 RMOVE -10,-10
```

Below is a listing of the program as it now looks:

```
100 INIT
110 SET DEGREES
120 PAGE
130 ROTATE 0
132 GOSUB 140
134 ROTATE 90
136 GOSUB 140
138 END
140 MOVE 65,50
150 REM   DRAW A TRIANGLE
155 RMOVE -10,-10
160 RDRAW 40,10
170 RDRAW -40,10
180 RDRAW 0,-20
190 HOME
200 RETURN
```



If the program is run again by entering RUN, the result (shown in the preceding diagram) is a triangle oriented like the first one in the previous example, "pointed to the right", and another triangle. This triangle is also oriented at right angles to the first one but the axis of rotation is now centered inside both triangles. This effective rotation point is still the end point of an absolute MOVE command. The reason why the rotation axis changed was that the inserted RMOVE command at line 155 shifted the triangle in relation to the end point of the last absolute MOVE. This rotation axis can also be determined by the end point of an absolute DRAW.

It should be emphasized again that the ROTATE statement affects only relative moves and draws, that is, only the RDRAW and RMOVE commands. Some uses of the ROTATE command are discussed later in this manual.

Graphing Arrays

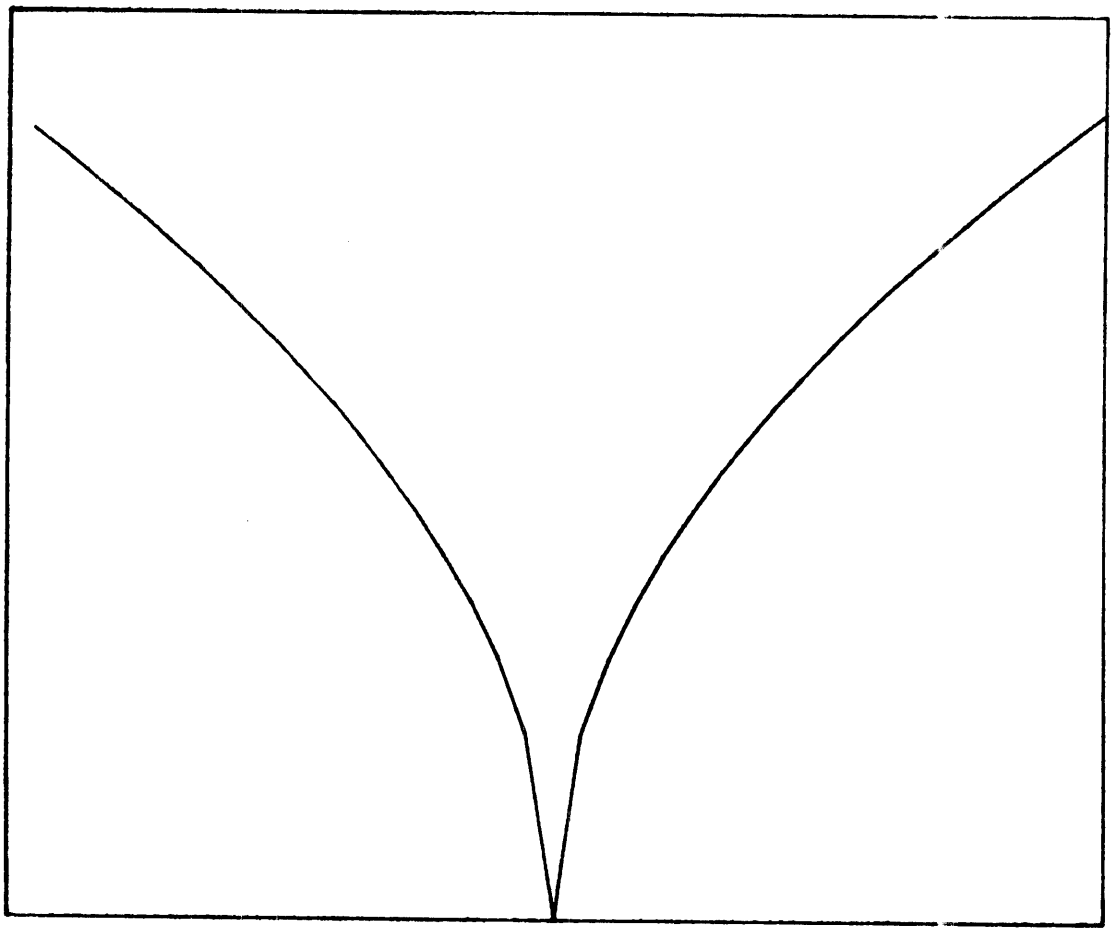
The MOVE, RMOVE, DRAW, and RDRAW commands all have a similar array output capability which greatly simplifies outputting a series of connected lines or other large amounts of graphic information. If forty lines are to be drawn on the screen, two one-dimensional arrays of length 40 are dimensioned as follows: DIM X(40),Y(40). Each successive X and Y value is placed in the corresponding array element by using a FOR . . . NEXT loop or any other appropriate process. In other words, the horizontal location (in user data units) of the first point is placed into X(1), the vertical location of the first point is placed into Y(1) and so forth until the arrays are filled. To draw these lines, all that is necessary is the command DRAW X,Y. This command will cause a line to be drawn from the current cursor location to the X(1),Y(1). From there it draws a line to X(2),Y(2). This continues until both arrays are exhausted. The arguments of the DRAW command can be either scales or arrays. The command will automatically handle either situation. Of course, both arguments must be scalars or both must be arrays. The types cannot be mixed.

If the variables X and Y are arrays, DRAW X,Y is the equivalent to:

```
310 FOR I=1 TO 40
320 DRAW X(I),Y(I)
330 NEXT I
```

This implied output sequence is applicable to DRAW, RDRAW, MOVE, and RMOVE. Here is an example:

```
PAGE
DELETE ALL
100 PAGE
110 INIT
120 DIM X(40),Y(40)
130 REM  FULL ARRAYS WITH DATA
140 FOR I=1 TO 40
150 X(I)=I*3.25
160 Y(I)=SQR(ABS(I-20))*20
170 NEXT I
180 REM  MOVE TO THE FIRST POINT
190 MOVE X(1),Y(1)
200 REM  DRAW BOTH ARRAYS
210 DRAW X,Y
220 HOME
230 END

RUN
```



Both arrays used must have the same number of elements. If they do not, an error condition results. Note also that this alternating output of array elements is different from that of the PRINT statement. If the statement PRINT X,Y is executed with the arrays X and Y dimensioned as above, the sequence of output is as follows: X(1),X(2),X(3), . . . X(39),X(40), Y(1),Y(2),Y(3), . . .Y(39),Y(40). All elements of the X array are output before the first element of the Y array is output.

WINDOW and VIEWPORT Examples

As a review, here are seven examples which illustrate capabilities of WINDOW and VIEWPORT. All seven examples use the same figure for output: a circle of radius 75 centered at X = 0 and Y = 0. They demonstrate how WINDOW and VIEWPORT determine what is shown on the display. Each example program performs three functions:

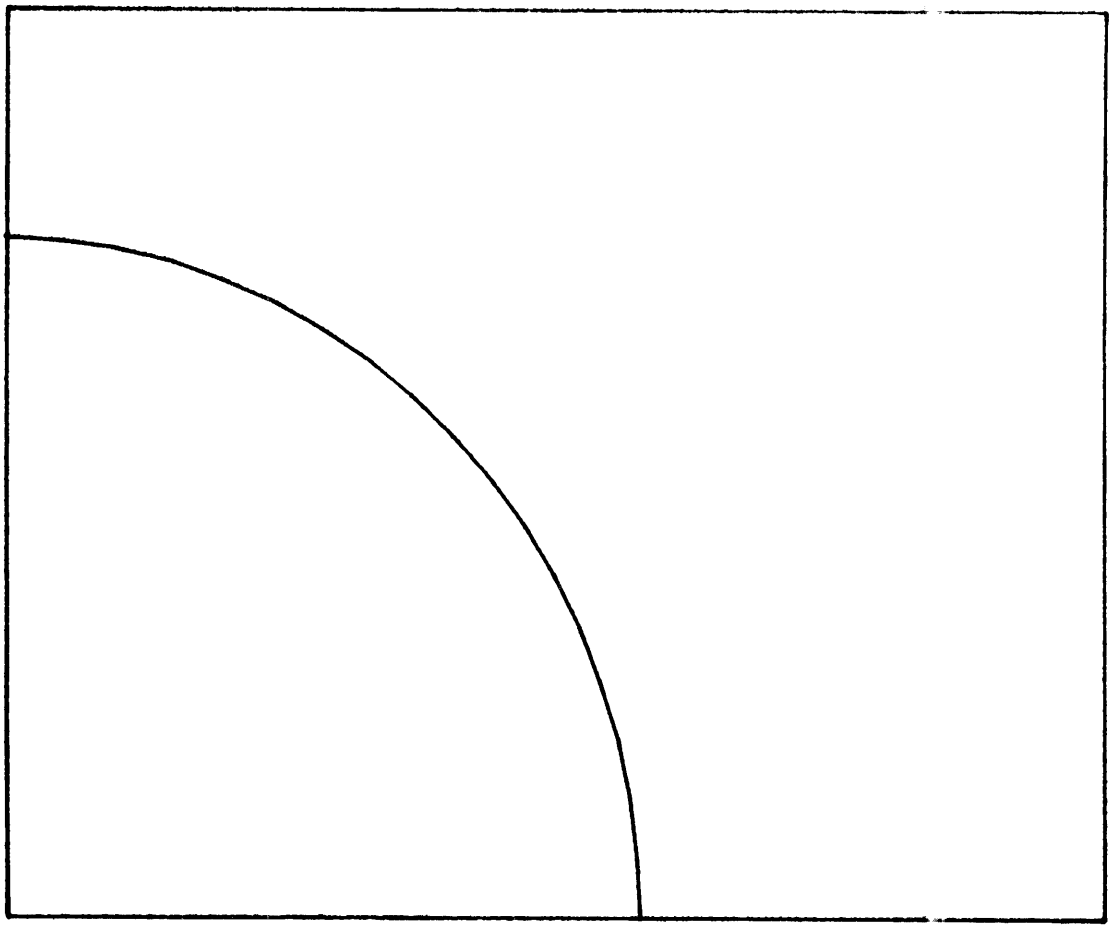
1. Fill arrays X and Y with data necessary to draw the circle
2. Define a window and viewport
3. Draw the circle.

In all seven examples, the statements which perform the first and third functions (above) are identical. The only difference among the programs is in function 2 — the window and viewport definition.

Please enter the following statements (this is the first of the seven examples):

```
PAGE
DELETE ALL
100 REM  FILL ARRAYS WITH DATA
110 REM  REPRESENTING A CIRCLE
120 INIT
130 DIM X(72),Y(72)
140 SET DEGREES
150 FOR I=1 TO 72
160 X(I)=75*COS(I*5)
170 Y(I)=75*SIN(I*5)
180 NEXT I
190 REM
200 PAGE
210 REM  DEFINE WINDOW AND VIEWPORT
220 WINDOW 0,130,0,100
230 VIEWPORT 0,130,0,100
240 REM   DRAW THE CIRCLE
250 GOSUB 500
260 HOME
270 END
500 REM  SUBROUTINE TO DRAW CIRCLE
510 MOVE X(72),Y(72)
520 DRAW X,Y
530 RETURN

RUN
```

In all seven examples, statements 100 through 190 are identical. These statements fill arrays X and Y with the appropriate data. Statements 500 through 530 are also identical in all seven examples. These statements form a subroutine to draw the data in arrays X and Y. The MOVE command at statement 510 positions the graphic point to the correct starting point for the circle to be drawn. This program's output is shown below:

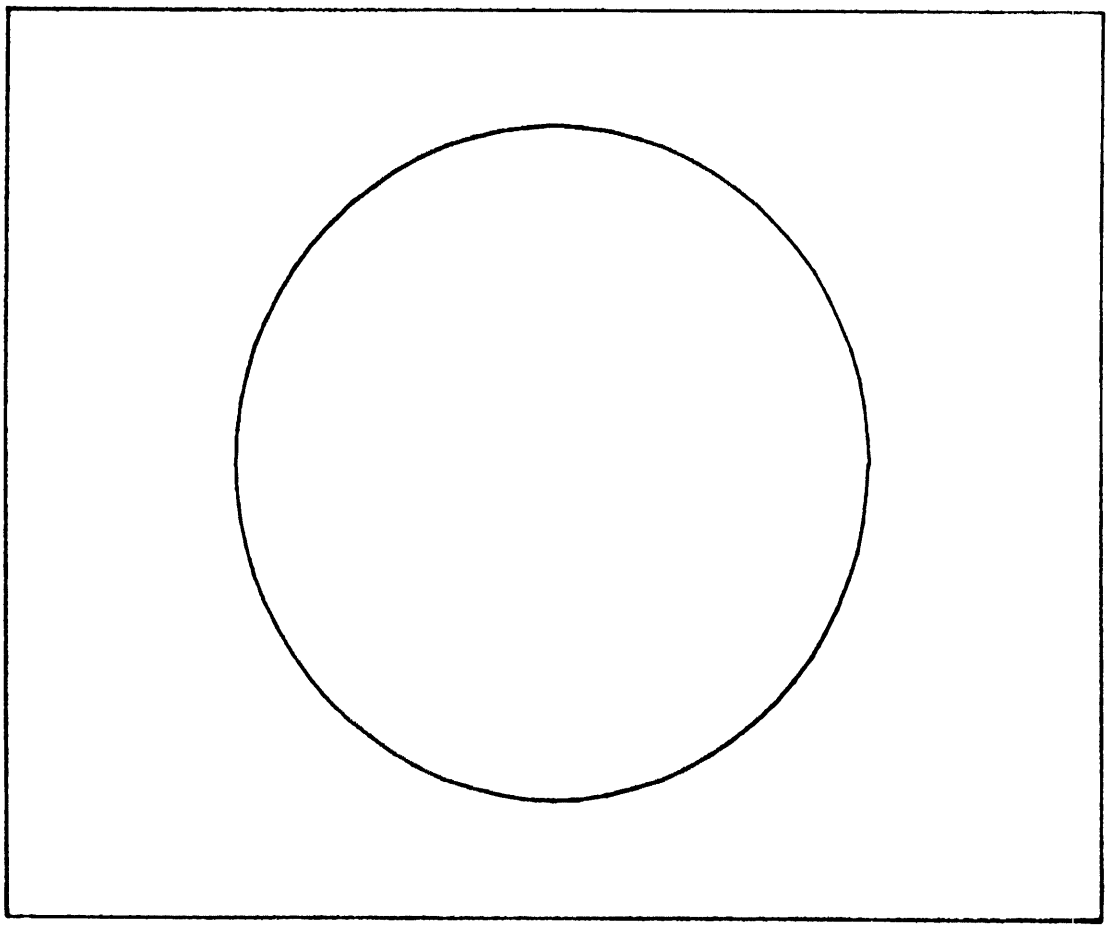


As a result of these window limits, only the upper right quarter of the circle is shown on the display.

In the example below, the window is changed so that the whole circle is visible. Enter the following statements:

```
220 WINDOW -130,130,-100,100
RUN 200
```

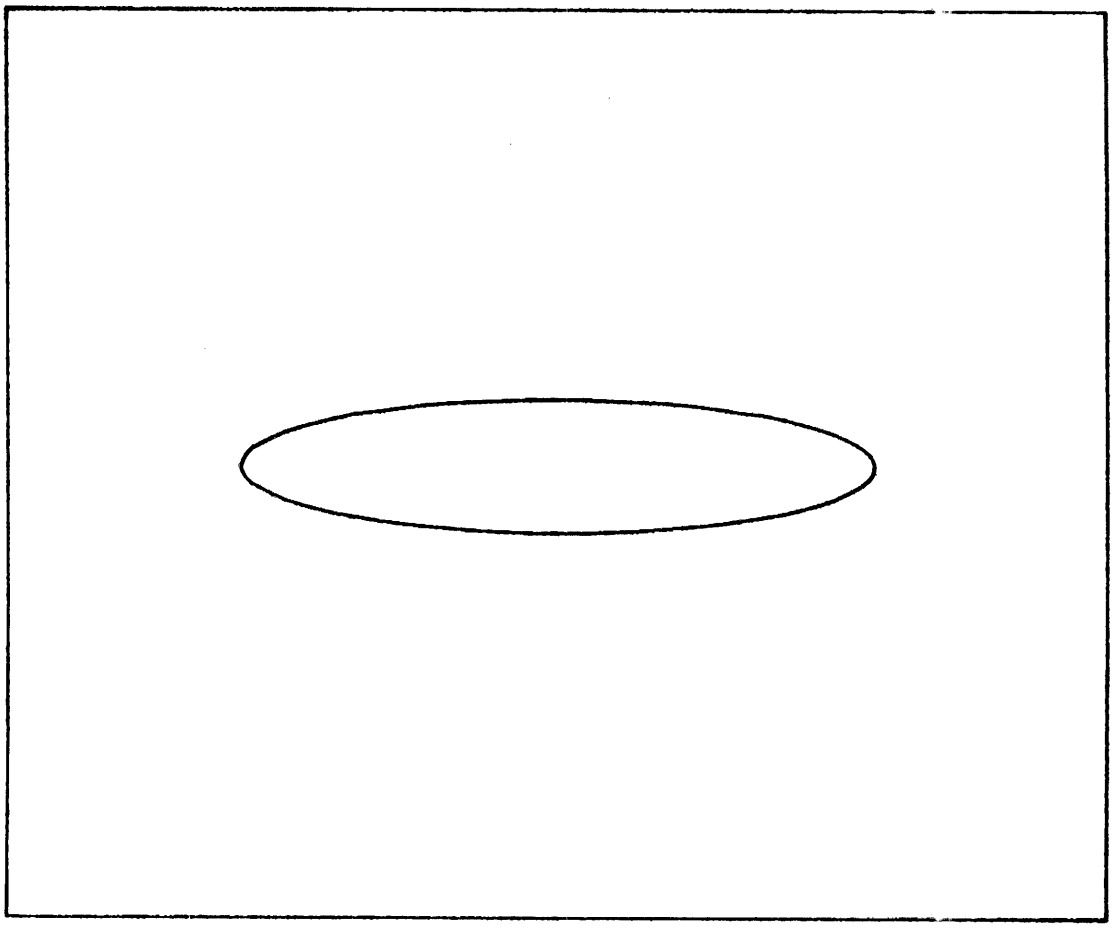
This produces the output shown below:



In the preceding example, the program was executed with the following command: RUN 200. (This causes the program execution to start at statement 200.) Once data arrays X and Y have been filled with data, as they were in the first example of this series, they need not be filled again. Since filling the arrays with data is the function of statements 100 through 190, these statements are skipped.

There is an implied horizontal and vertical scaling whenever a window is defined. The implied horizontal scaling and implied vertical scaling can be varied independently. The next two examples show that data which represents a circle will be drawn with a different shape if the window is defined inappropriately. Enter the following statements:

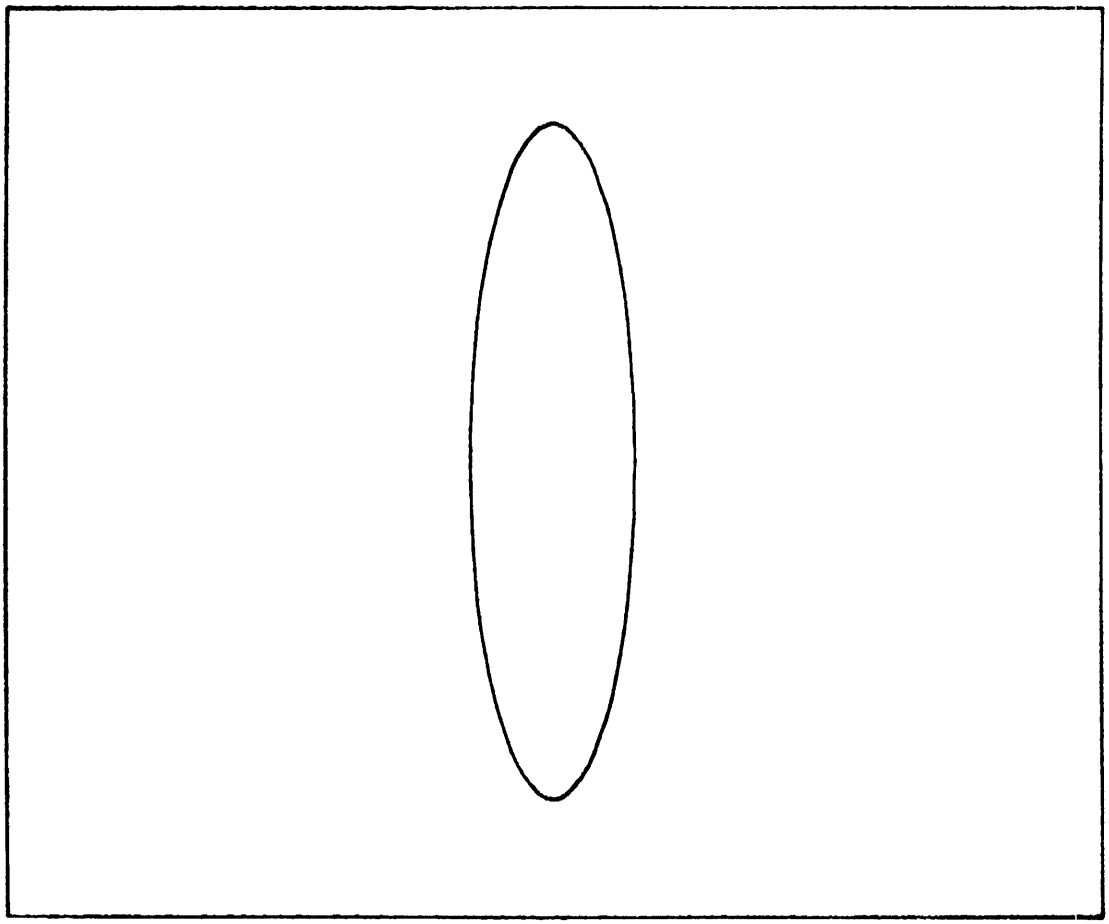
```
220 WINDOW -130,130,-500,500
RUN 200
```



In the program output above, the width of the drawn figure is the same as in the example which immediately preceded it. The WINDOW command arguments which specify the horizontal data range are identical in both examples. However, the WINDOW command arguments which specify the vertical data range are different. This causes the different shaped figures.

The next example changes the horizontal range. Enter the following:

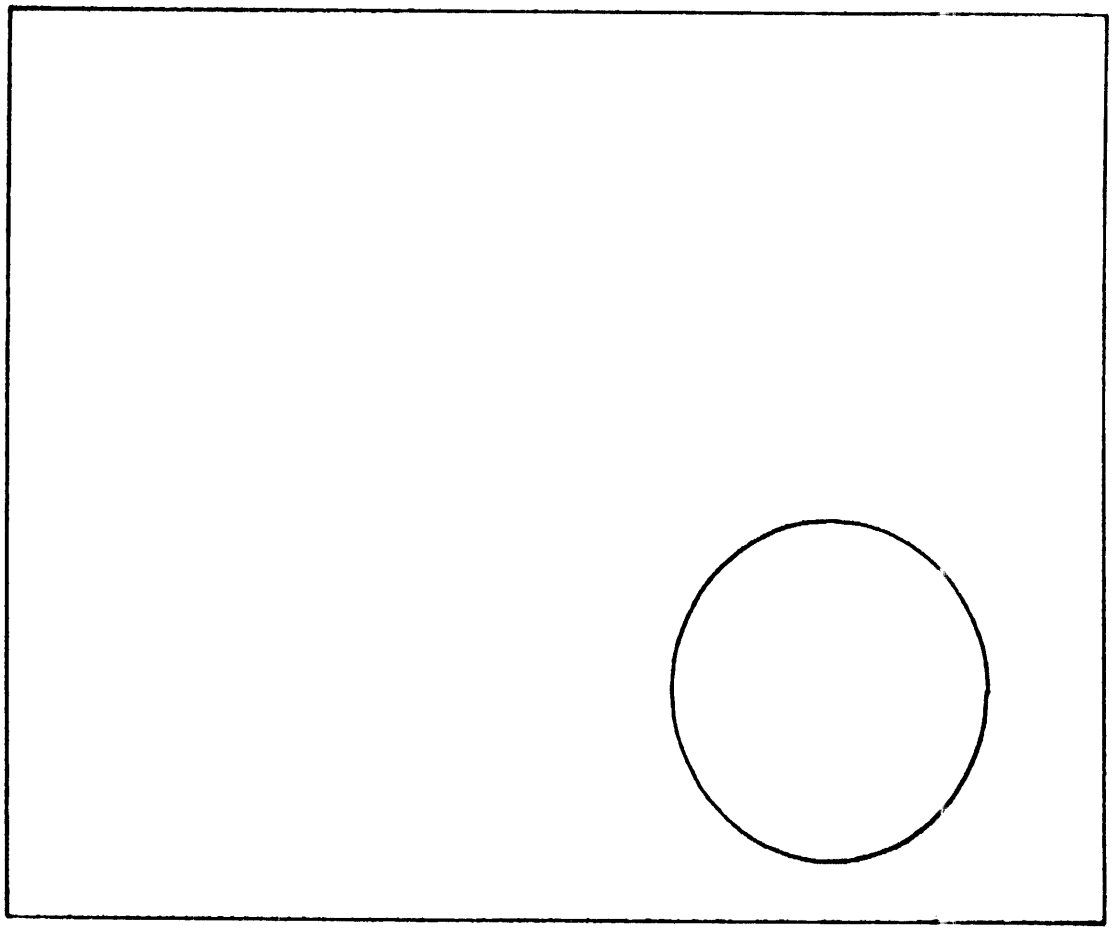
```
220 WINDOW -500,500,-100,100
RUN 200
```



To draw a figure with no distortion, the ratio of the viewport's height and width must be the same as the ratio of the window's height and width. This is discussed in more detail in Section 8.

In the next example the window is restored to its previous dimensions (−130,130,−100,100) but the viewport is changed so that it no longer coincides with the full size of the display. Enter the following:

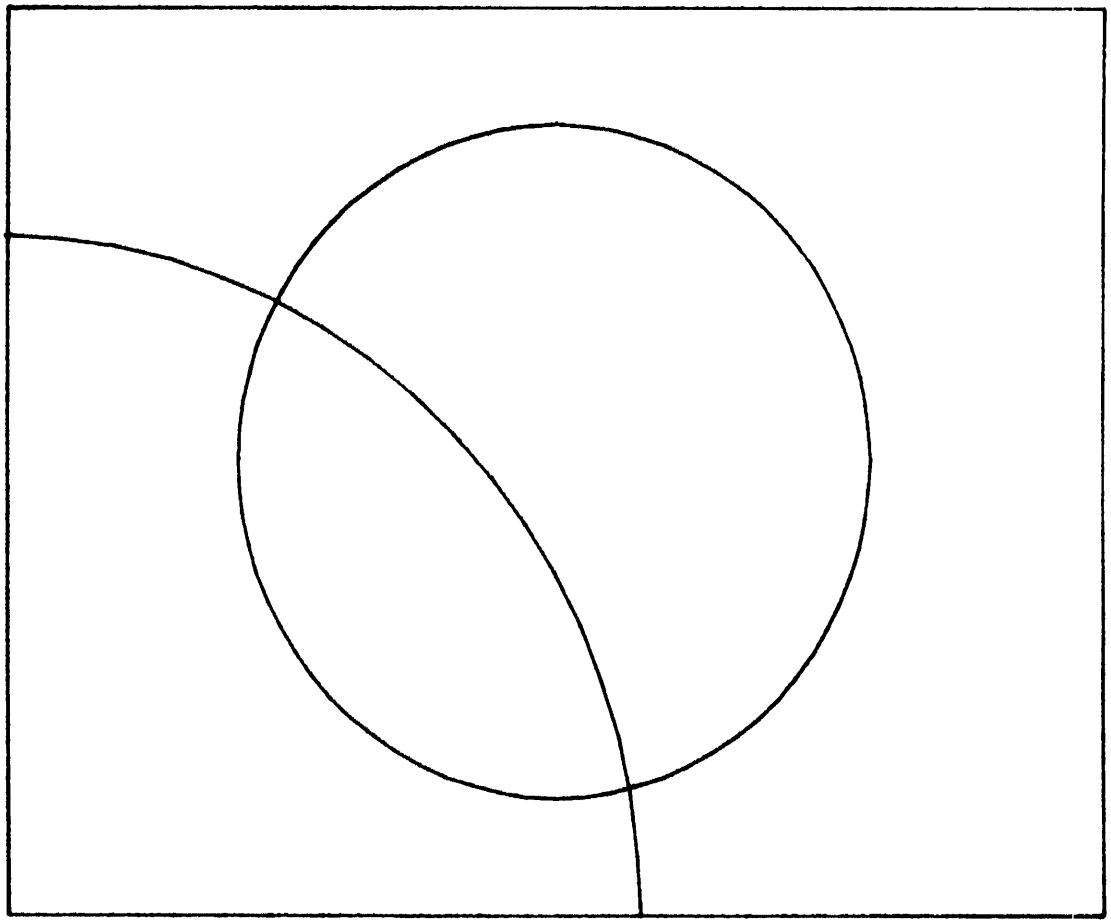
```
220 WINDOW -130,130,-100,100
230 VIEWPORT 65,130,0,50
RUN 200
```



As the program output shows, the viewport is now defined to be the lower right corner of the display. This was specified in the VIEWPORT command in statement 230. There has been an implied change of scale because the same window must fit into a smaller portion of the display.

The next example shows how different windows can be defined for the same display area. Enter the following statements:

```
PAGE
220 WINDOW 0,130,0,100
230 VIEWPORT 0,130,0,100
240 GOSUB 500
250 WINDOW -130,130,-100,100
260 GOSUB 500
270 HOME
280 END

RUN200
```



The command GOSUB 500 appears twice. As a result, the circle is drawn twice. There are two WINDOW commands. Each causes a different portion of the circle to be visible when the routine at statement 500 is called. However, only one viewport is defined (in statement 230). This means that both windows occupy the same area on the display.

In the previous example, statements 220 through 260 determine what is placed on the display. Here is a summary of their functions:

220 WINDOW 0,130,0,100	Specifies a window which covers the upper right section of the circle.
230 VIEWPORT 0,130,0,100	Specifies a viewport which fills the display.
240 GOSUB 500	Draws the circle subject to the defined window and viewport. Because of the WINDOW command at statement 220, only the upper right section of the circle appears on the display.
250 WINDOW −130,130,−100,100	Specifies a window which covers the whole circle.
260 GOSUB 500	Draws the circle subject to the defined window and viewport. Because of the WINDOW command at statement 250, the complete circle appears on the display.

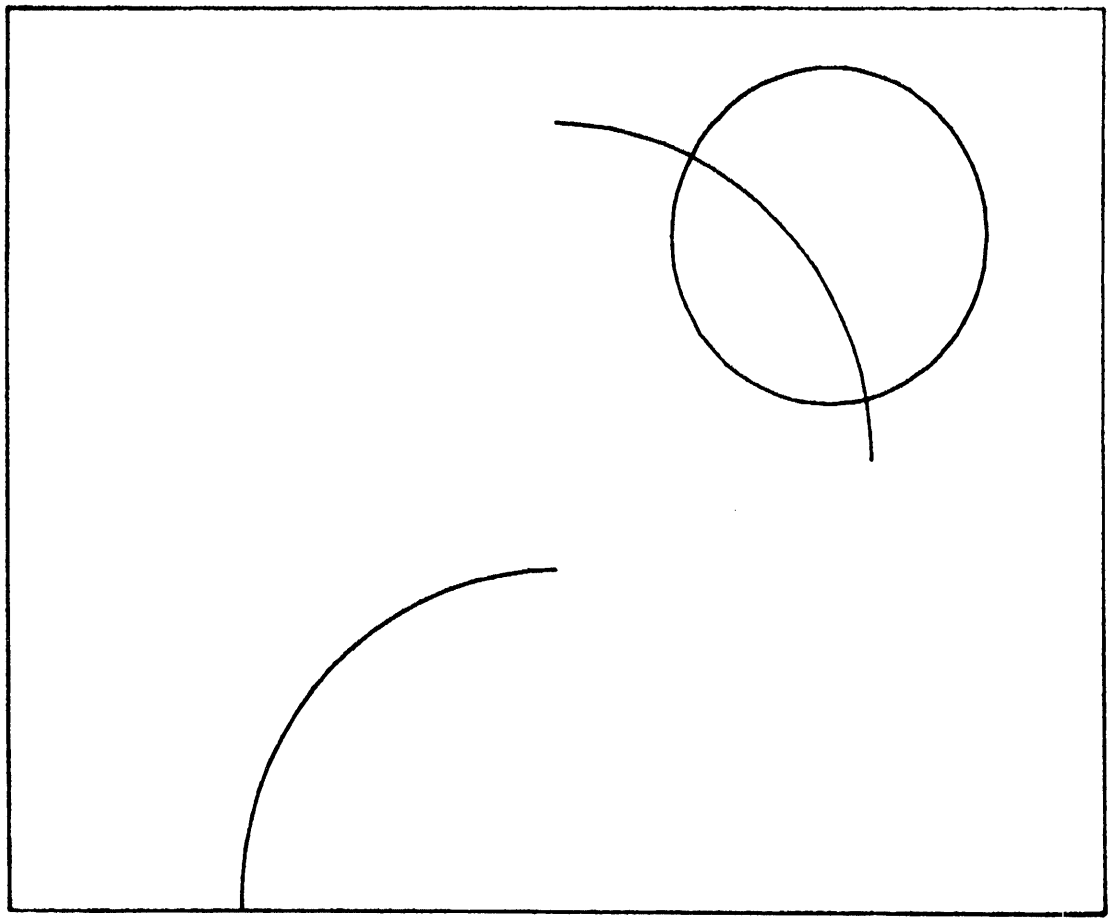
The next example draws everything seen on the display in the above example but, through a change in viewport parameters, draws the image in the upper right quarter of the display. It also adds some additional data in the lower left quarter of the display. Enter the following statements:

```
PAGE
230 VIEWPORT 65,130,50,100
270 WINDOW -130,0,0,100
280 VIEWPORT 0,65,0,50
290 GOSUB 500
300 HOME
310 END

RUN200
```

The entire program now looks like this:

```
100 REM  FILL ARRAYS WITH DATA
110 REM  REPRESENTING A CIRCLE
120 INIT
130 DIM X(72),Y(72)
140 SET DEGREES
150 FOR I=1 TO 72
160 X(I)=75*COS(I*5)
170 Y(I)=75*SIN(I*5)
180 NEXT I
190 REM
200 PAGE
210 REM  DEFINE WINDOW AND VIEWPORT
220 WINDOW 0,130,0,100
230 VIEWPORT 65,130,50,100
240 GOSUB 500
250 WINDOW -130,130,-100,100
260 GOSUB 500
270 WINDOW -130,0,0,100
280 VIEWPORT 0,65,0,50
290 GOSUB 500
300 HOME
310 END
500 REM  SUBROUTINE TO DRAW CIRCLE
510 MOVE X(72),Y(72)
520 DRAW X,Y
530 RETURN
```

The output from this program looks like this:



The above example demonstrates that any portion of the user data space can be shown at any place on the display. Statements 220 through 290 determine what is visible on the display. Here is a summary of their functions:

220 WINDOW 0,130,0,100	Specifies a window which covers the upper right section of the circle.
230 VIEWPORT 65,130,50,100	Specifies a viewport to be the upper right quarter of the display.
240 GOSUB 500	Draws the circle subject to the defined viewport and window. Because of the WINDOW command at statement 220, only the upper right section of the circle appears on the display.
250 WINDOW −130,130,−100,100	Specifies a window which covers the whole circle.
260 GOSUB 500	Draws the circle subject to the defined window and viewport. The complete circle appears because of the WINDOW command at statement 250.

270 WINDOW –130,0,0,100	Specifies a window which covers the upper left section of the circle.
280 VIEWPORT 0,65,0,50	Specifies a viewport to be the lower left quarter of the display.
290 GOSUB 500	Draws the circle subject to the defined window and viewport. Because of the WINDOW command at statement 270, only the upper left section of the circle appears.

WINDOW and VIEWPORT Summary

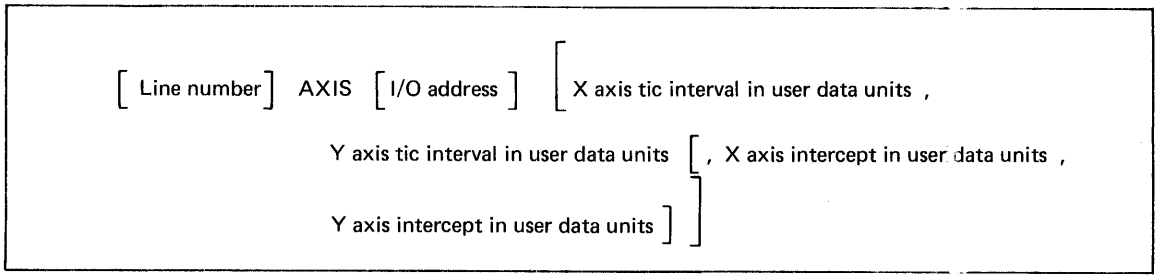
All seven examples illustrate the same two principles:

- The window and viewport are defined independently. Different windows can be defined for one viewport; different viewports can be filled with the same window.

- The currently defined window and viewport together determine what appears on the display.

WINDOW defines what data is seen. VIEWPORT defines where on the display it appears.

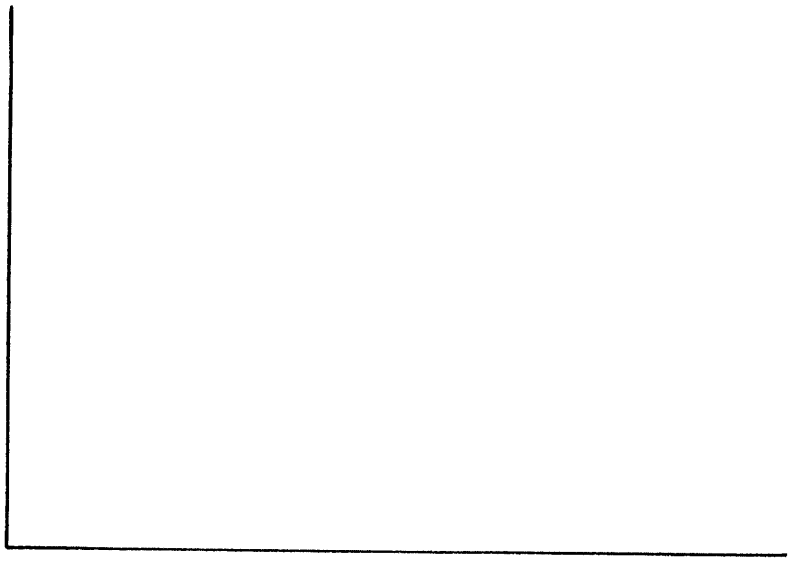
AXIS Command



The AXIS command draws horizontal and vertical axis lines, with tic marks if desired. The command can be used with four arguments, with two arguments or with no arguments. Default values are supplied in place of any missing arguments. The AXIS command with no arguments draws a horizontal and a vertical line through the origin of the user data space (X = 0 and Y = 0 in user data units) if this point is inside the defined window. If this point is not inside the window, the axis lines are drawn through the minimum X and Y data values. In other words, unless X = 0 is within the window, the vertical or Y axis is drawn at the left edge of the window; and unless Y = 0 is within the window the horizontal or X axis is drawn at the bottom edge of the window.

The following examples illustrate the default location of the axes. Enter the following:

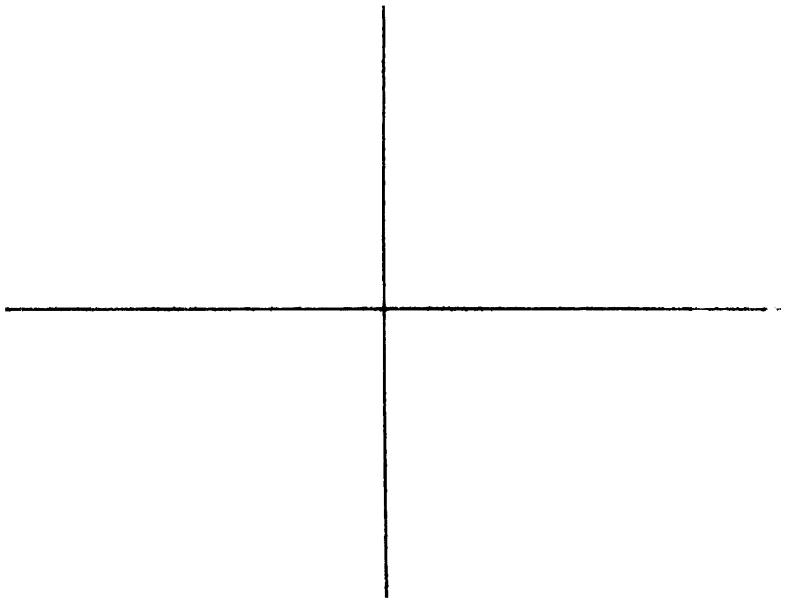
```
DELETE ALL
PAGE
100 PAGE
110 INIT
120 AXIS
130 END

RUN
```



Since INIT is equivalent to WINDOW 0,130,0,100, the point X = 0 and Y = 0 is at the lower left corner of the display. So the vertical axis is at the left edge of the display and the horizontal axis is at the bottom edge. After the AXIS is drawn, the graphic point is placed at the intersection of the two axes. In the above example, the two axes intersect at the lower left corner of the display. When execution is complete, that is the cursor's location.

Press the PAGE key and enter the following statements:

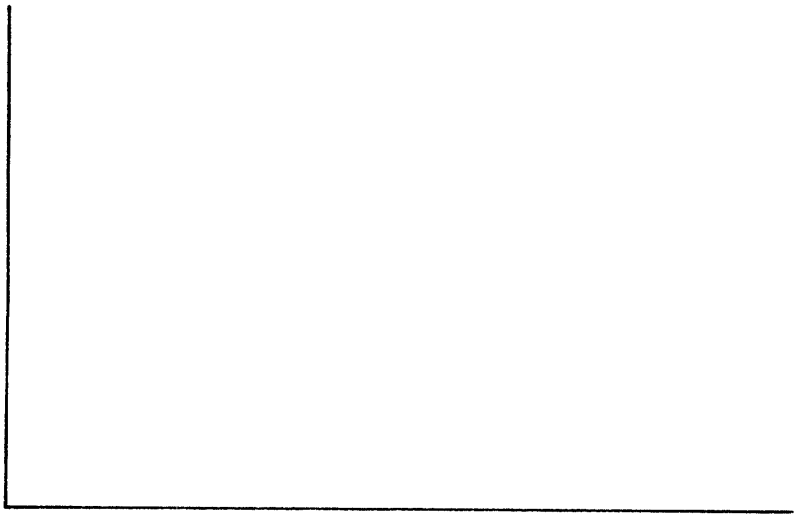
```
115 WINDOW -50,50,-50,50
125 HOME
RUN
```



The axis lines cross at the center of the display because that is the location of the origin (X = 0 and Y = 0) of the user data space. (The HOME command at statement 125 is added to make updating this program easier.)

Enter the following:

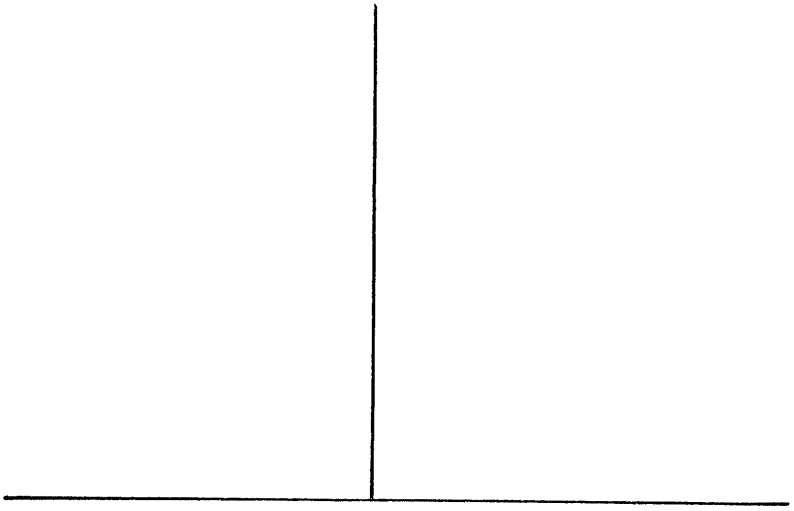
```
115 WINDOW -100,-50,-100,-50
RUN
```



The axis lines cross at the lower left corner of the screen, because that is where the minimum values of X and Y occur. Notice that the AXIS statement defines minimum value to be minimum algebraic value, not minimum absolute value.

Enter the following:

```
115 WINDOW -50,50,50,100
RUN
```

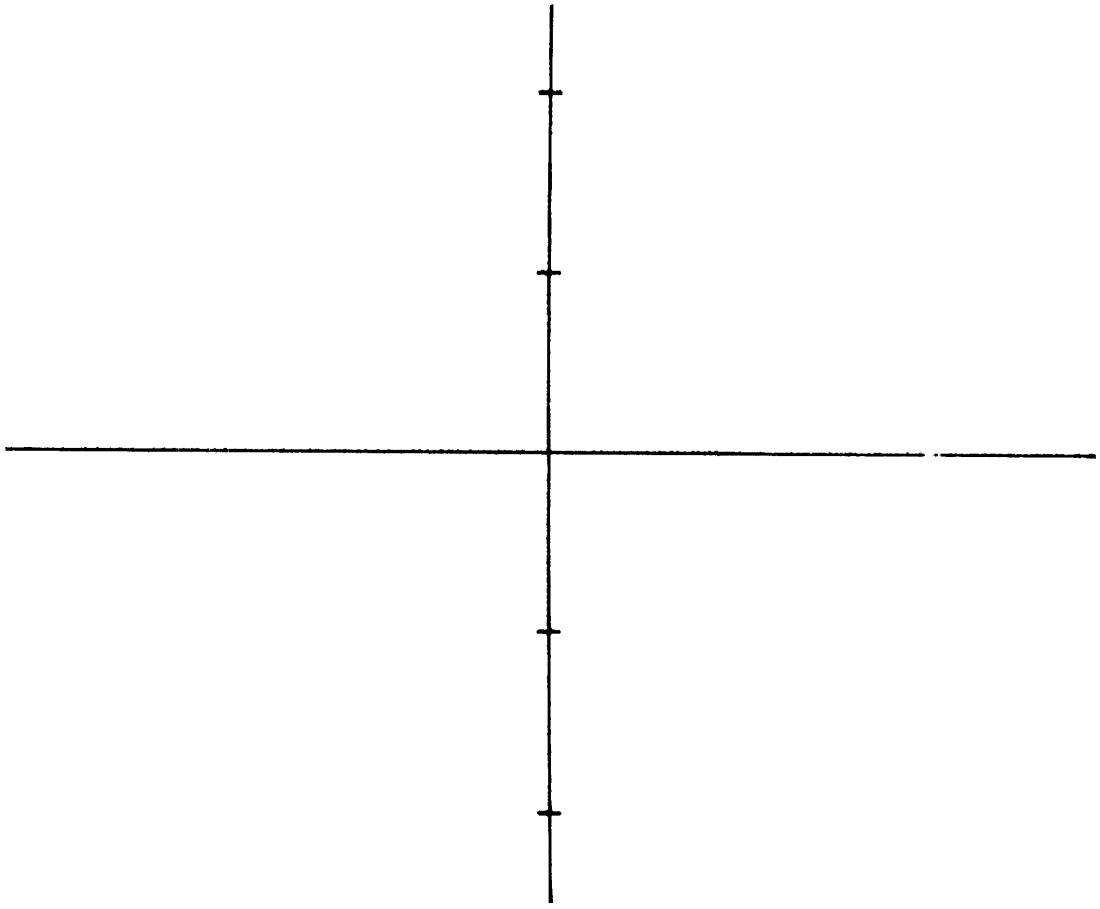


The vertical axis is now centered on the display because the point X = 0 is located halfway between the left and right edges of the display. The horizontal axis, as before, passes through the Y data minimum and, as a result, is located at the lower edge of the display.

The first two arguments of AXIS (whether it is being used with four arguments or only two) specify the interval in user data units between tic marks. The first argument is the interval between tics on the horizontal or X axis; the second argument is the interval between tics on the vertical or Y axis. If an interval of zero units is specified in either argument, no tics at all are drawn on the axis corresponding to that argument.

Enter the following:

```
115 WINDOW -25,25,-25,25
120 AXIS 0,10
RUN
```



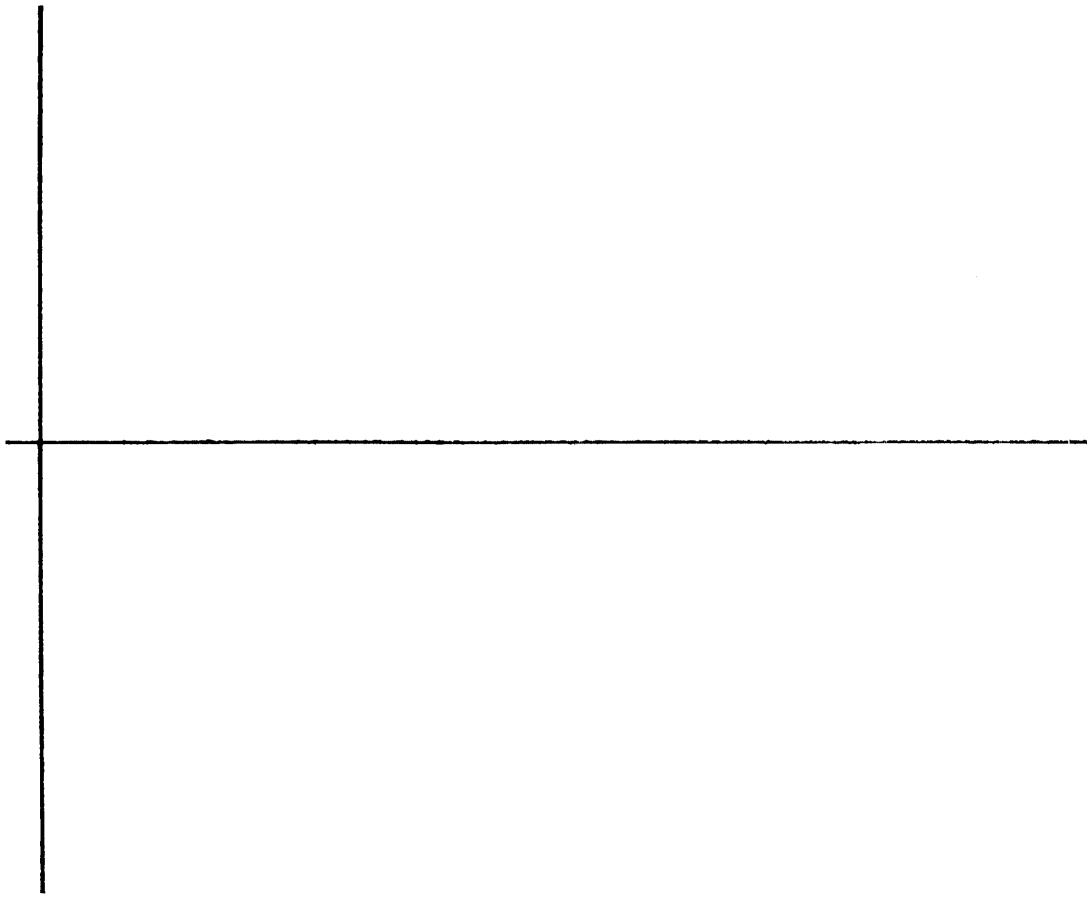
Because the arguments in the WINDOW statement have placed the point X = 0 and Y = 0 in the middle of the screen, the axes are drawn with their crossing point centered on the screen. Notice there are no tic marks on the horizontal axis (because the first argument of the AXIS

statement is zero). However, because the second argument of AXIS is 10, there are tic marks on the vertical axis spaced 10 user data units apart. The tic marks are always aligned with the crossing point of the axes, as they are here. The length of the tic marks is always 1% of the viewport size in the corresponding direction. For example, if the AXIS statement is executed after a VIEWPORT 0,50,0,100 statement, the tic marks on the vertical axis are .5 GDU in length (1% of the 50 GDU width of the defined viewport) and the tic marks on the vertical axis are 1 GDU in length (1% of the 100 GDU height of the defined viewport).

The third and fourth arguments of the AXIS statement allow the user to specify the horizontal and vertical intercepts of the axes. The third argument is the location in horizontal or X data units of the vertical axis intercept. Similarly, the fourth argument is the location in vertical or Y data units of the horizontal axis intercept.

Enter the following:

```
120 WINDOW 0,260,0,200
130 AXIS 0,0,10,100
RUN
```



The AXIS statement causes the vertical axis to be drawn through X = 10, a point near the left edge of the display, and causes the horizontal axis to be drawn through Y = 100, a point in the middle of the display.

Axis can be used to easily draw a box around the specified window. The example below illustrates how this is done. Press the PAGE key and enter the following statements:

```
DELETE ALL
100 PAGE
110 INIT
120 DATA -40,55,-22,89
130 READ W1,W2,W3,W4
140 WINDOW W1,W2,W3,W4
150 AXIS 0,0,W1,W3
160 AXIS 0,0,W2,W4
170 HOME
180 END

RUN
```

The AXIS command at statement 150 draws lines at the left and bottom edges of the window. For this particular task, two AXIS commands perform the function of one MOVE and four DRAW commands.

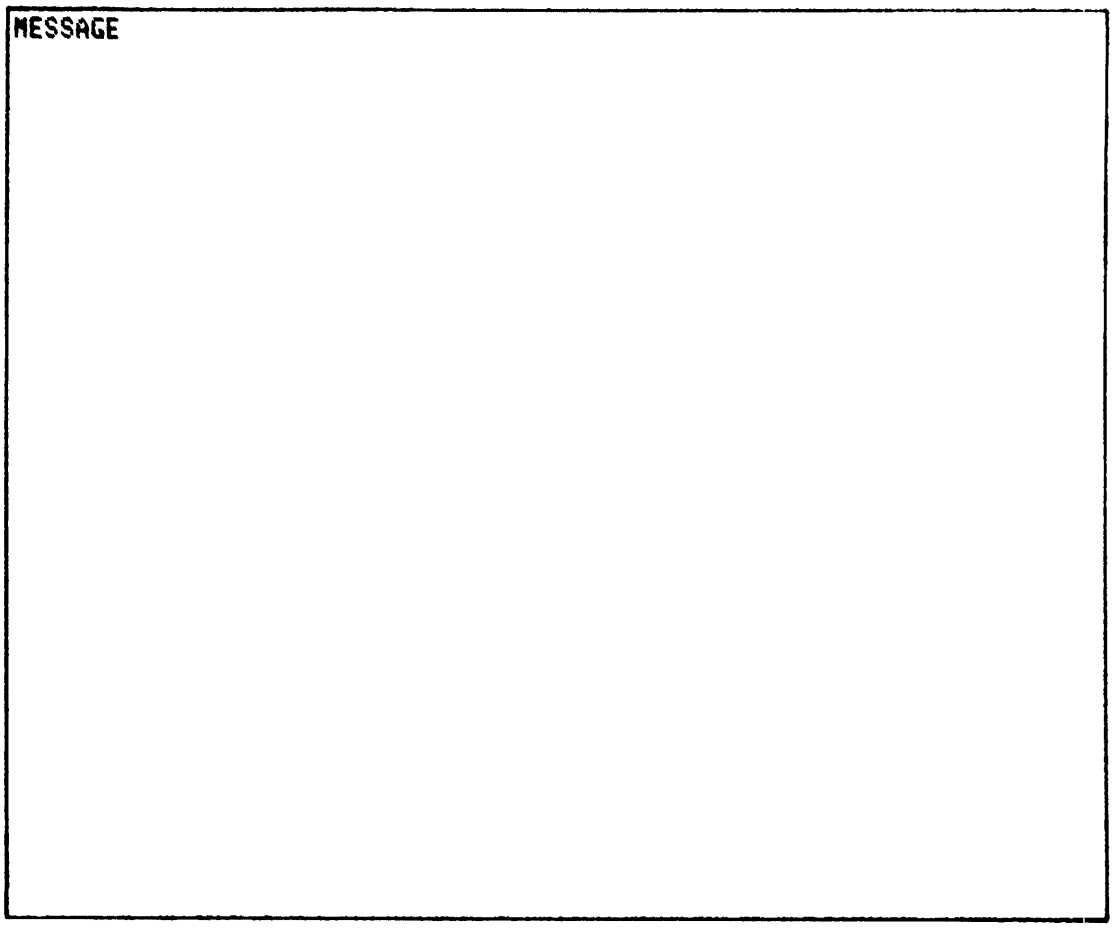
CHARACTER OUTPUT

This part of the manual discusses output of characters (letters and numbers) to a graphic device. This is done with the PRINT statement. A fundamental concept of outputting information to a graphic device is that the cursor's location defines the starting point for both character output and graphic output (lines). If a DRAW statement is executed, the starting point of the resulting line is the lower left corner of the cursor's current position. If a PRINT statement is executed, the first character is normally output at the cursor's current position and the cursor is moved to the position of the next character (to the right of the cursor's starting position).

Positioning with MOVE

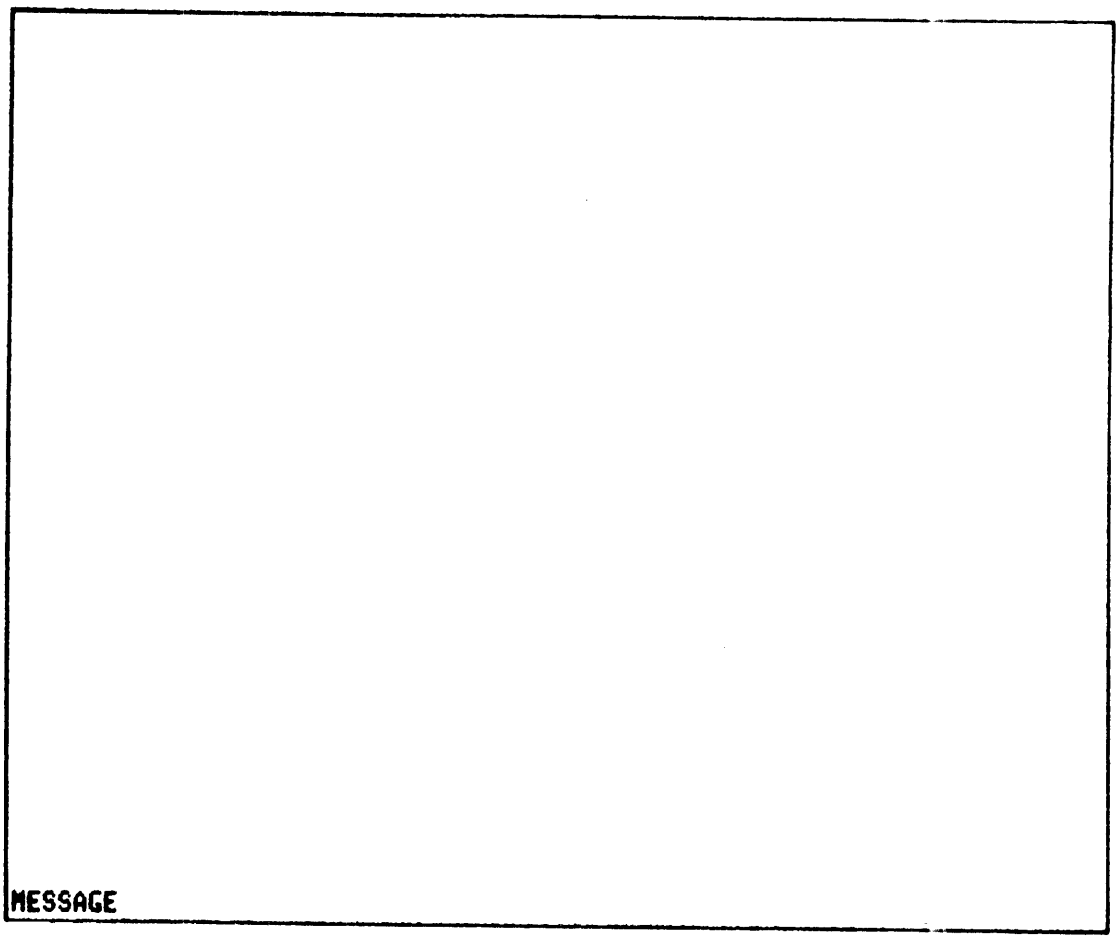
Character output can be placed anywhere on the display by preceding the PRINT with a MOVE. For example; press the PAGE and enter the following statements:

```
DELETE ALL
100 PAGE
110 INIT
120 PRINT "MESSAGE";
130 HOME
140 END
RUN
```



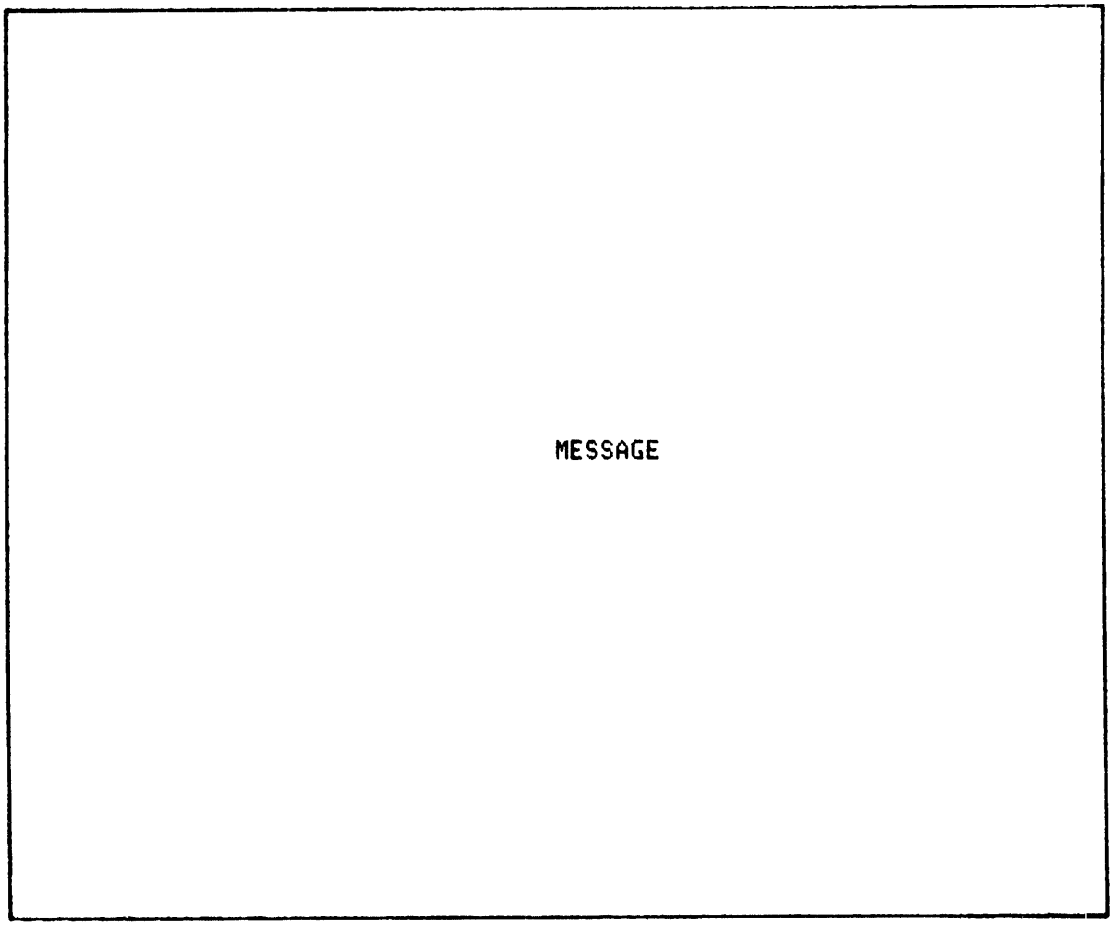
The message is placed at the upper left corner of the display because that is the cursor's position after the PAGE statement is executed. Enter the statements below:

```
115 MOVE 0,0
RUN
```



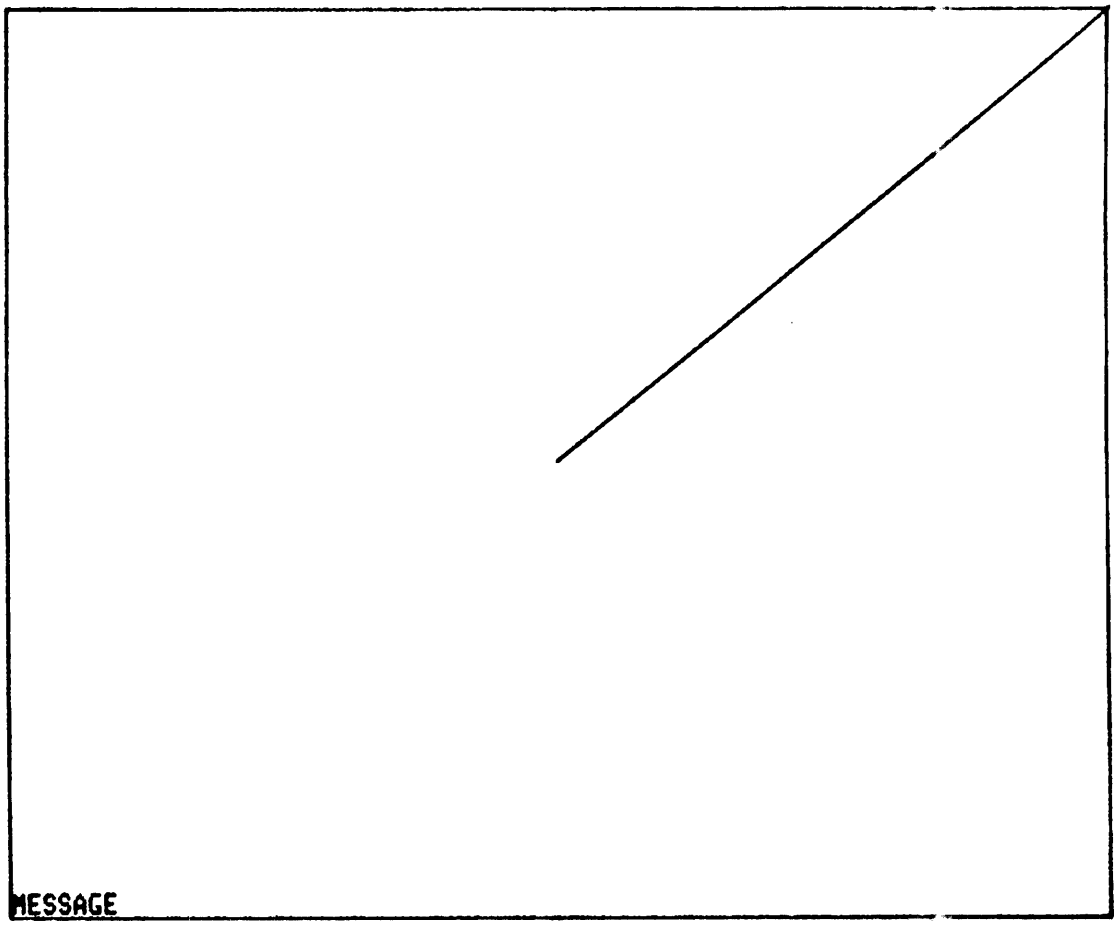
The message is placed in the lower left corner of the screen because of the MOVE 0,0 statement. Enter the following statements:

```
115 MOVE 65,50
RUN
```



In this case the message is placed in the middle of the screen because of the MOVE 65,50 which places the cursor in the center of the screen. Enter the following statements:

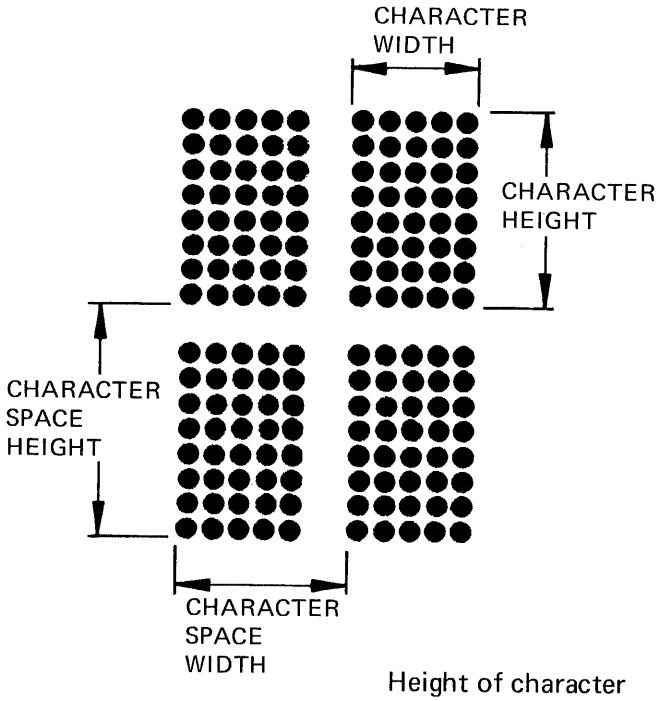
```
DELETE ALL
100 PAGE
110 WINDOW 0,130,0,100
120 VIEWPORT 65,130,50,100
130 MOVE 130,100
140 DRAW -130,-100
150 MOVE -130,-100
160 PRINT "MESSAGE";
170 HOME
180 END

RUN
```



Notice that, although the line resulting from the DRAW statement is clipped at the boundaries set up by VIEWPORT, the MOVE and PRINT statements are not clipped. After the WINDOW and VIEWPORT statements are executed, the center of the display is equivalent to X = 0 and Y = 0 in user data units. It is also the lower left corner of the viewport. The DRAW statement attempts to draw a line beyond this point but is prevented from doing so. However, the MOVE statement can place the cursor at any point on the display and the PRINT statement can place characters at any point on the display.

Character Size

Characters placed on an 11-inch display with a PRINT command have only one size and orientation. The size of the characters has four components: the height of the five by eight rectangular dot matrix which is used to form the character, the width of that matrix, the horizontal distance between adjacent characters in a line of characters, and the vertical distance between adjacent lines of characters. Refer to the figure below. These distances on an 11-inch display are all constant and are expressed in GDU's as follows:



Height of character	1.88 GDU's
Width of character	1.55 GDU's
Character space width	1.79 GDU's
Character space height	2.82 GDU's

Positioning with PRINT

As described previously, the cursor can be moved to any point on the display by using the MOVE statement. There are also four characters which move the cursor on the display horizontally in increments of the distance between adjacent characters and vertically in increments of the distance between adjacent lines. Using these characters, the cursor can be moved in any direction and thus can also be moved to any point on the display.

The following chart summarizes these four characters:

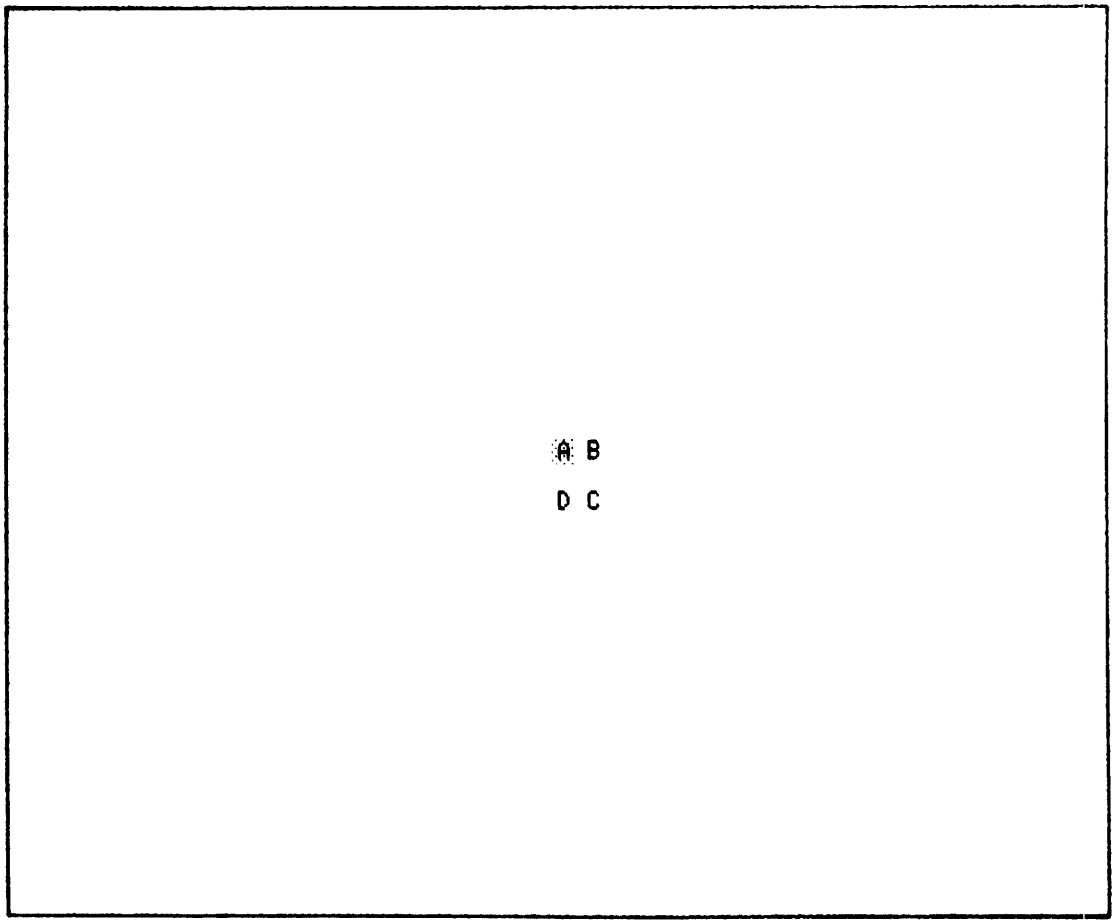
Action Moves Cursor	Character Name	Obtainable From Keyboard With	How to Place Into a Character String	ASCII Value
To the right	SPACE	SPACE BAR	A$ = " "	32
To the left	BACKSPACE	CTRL H	A$ = "H"	8
Down	LINE FEED	CTRL J	A$ = "J"	10
Up	VERTICAL TAB	CTRL K	A$ = "K"	11

Three of these characters are entered as control characters: Control H, Control J, and Control K.

To enter a control character into the Graphic System, the CTRL key is used much like the SHIFT key. A control character is entered by pressing the desired character key while simultaneously pressing and holding the CTRL key. The "space" character (entered with the SPACE BAR) moves the cursor to the right by 1.79 GDU's, the distance which separates adjacent characters. The "backspace" character (which prints on the screen as "H" or "Control H") moves the cursor to the left by the same distance. The "linefeed" character (which prints on the screen as "J" meaning "Control J") moves the cursor down by 2.82 GDU's, the vertical distance which separates adjacent lines of characters. The "vertical tab" character (which prints on the screen as "K" or "Control K") moves the cursor up by the same distance. These distances are all GDU's and are not affected by the WINDOW, VIEWPORT, or SCALE statements. When manipulating the cursor in this way, it must be remembered that the cursor is moved to the right whenever a non-control character is output. For an example of this method of cursor manipulation, press the PAGE and enter the following:

```
DELETE ALL
100 INIT
110 PAGE
120 MOVE 65,50
130 PRINT "A BHJJCHHHDHKK";
140 END

RUN
```

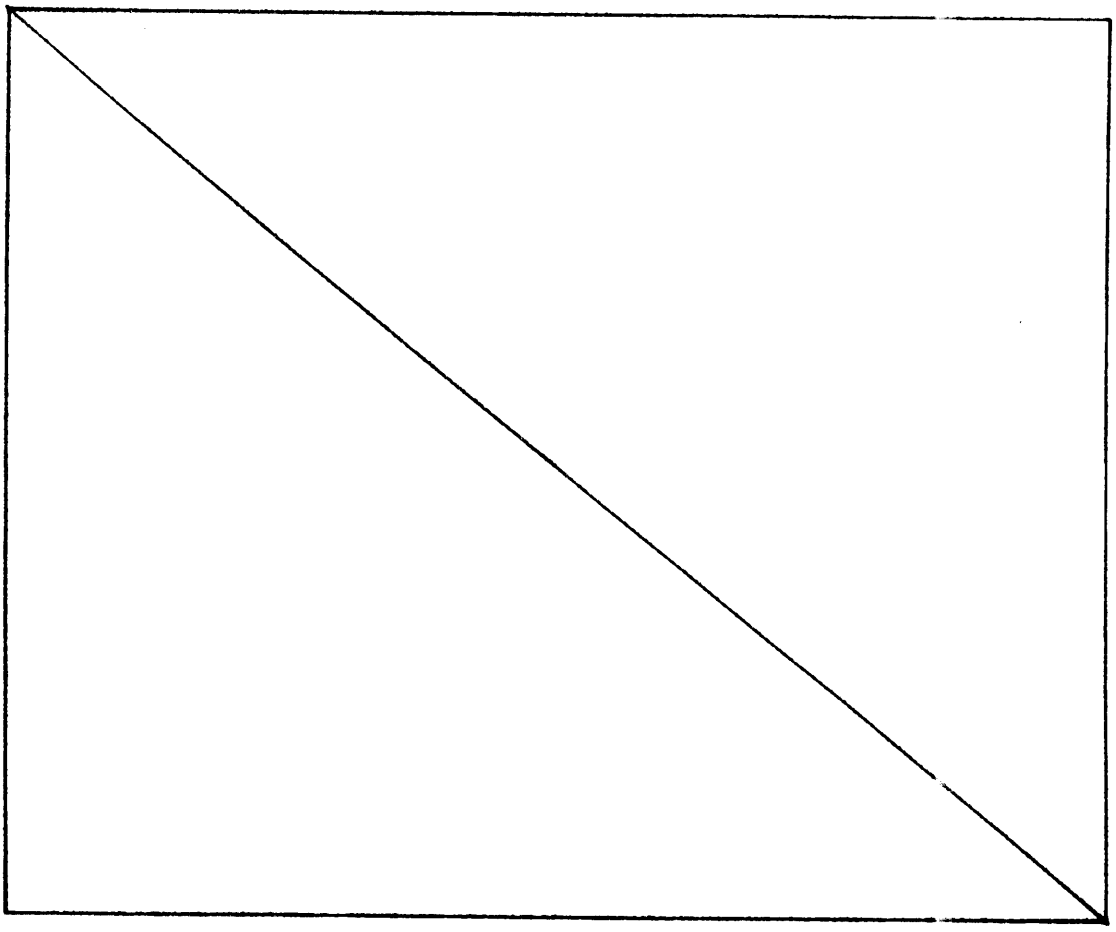
The resulting output looks like this:



When the program has finished execution, the cursor is placed in the same position as the first character output (the A).

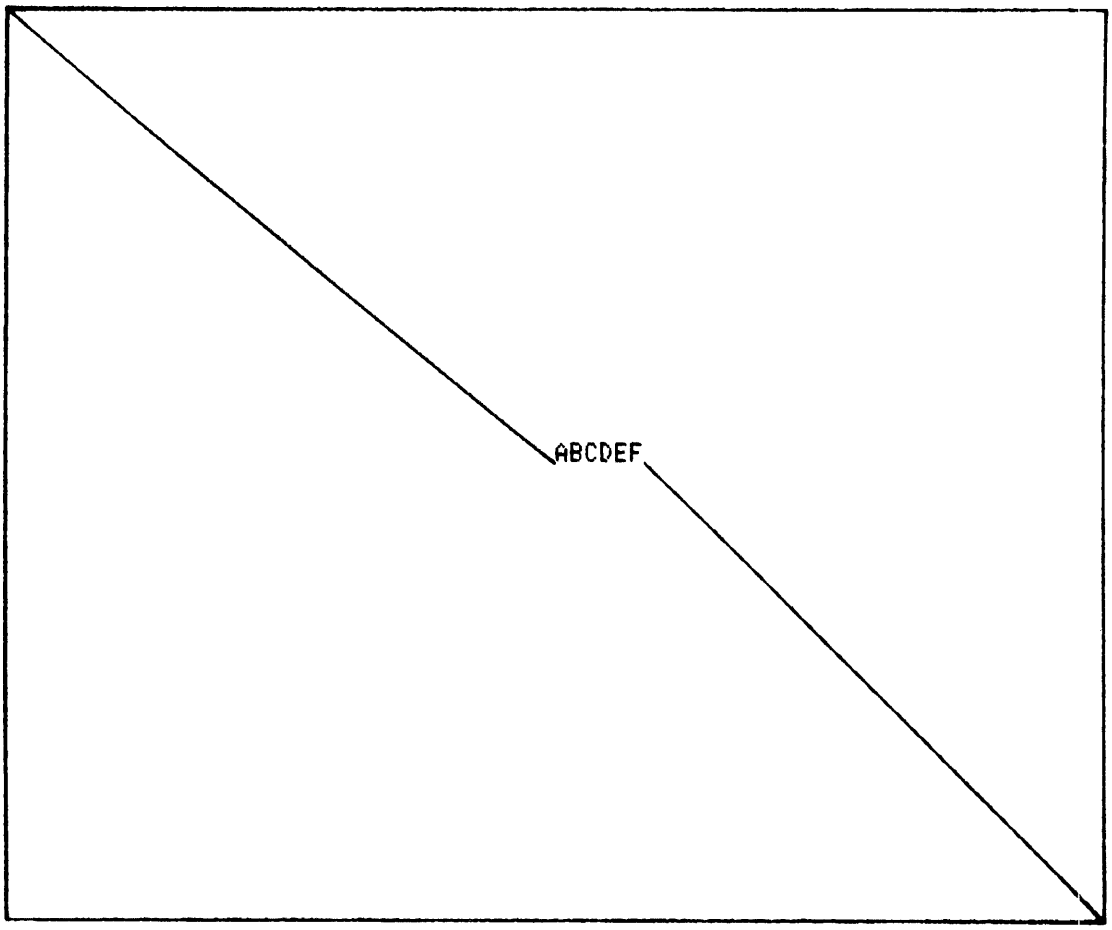
As mentioned earlier, a fundamental point about outputting information to a graphic device is that the cursor's location defines the starting point for both character output and graphic output. For example, here is a program which draws two connected lines from the upper left corner of the display to the lower right corner of the display. Press PAGE key and enter the following statements:

```
DELETE ALL
100 PAGE
110 INIT
120 MOVE 0,100
130 DRAW 65,50
140 DRAW 130,0
150 HOME
160 END

RUN
```



If a PRINT statement is added between the two drawn lines, the end point of the first line and the beginning point of the second line no longer coincide. Enter the following statements:

```
135 PRINT "ABCDEF";
RUN
```



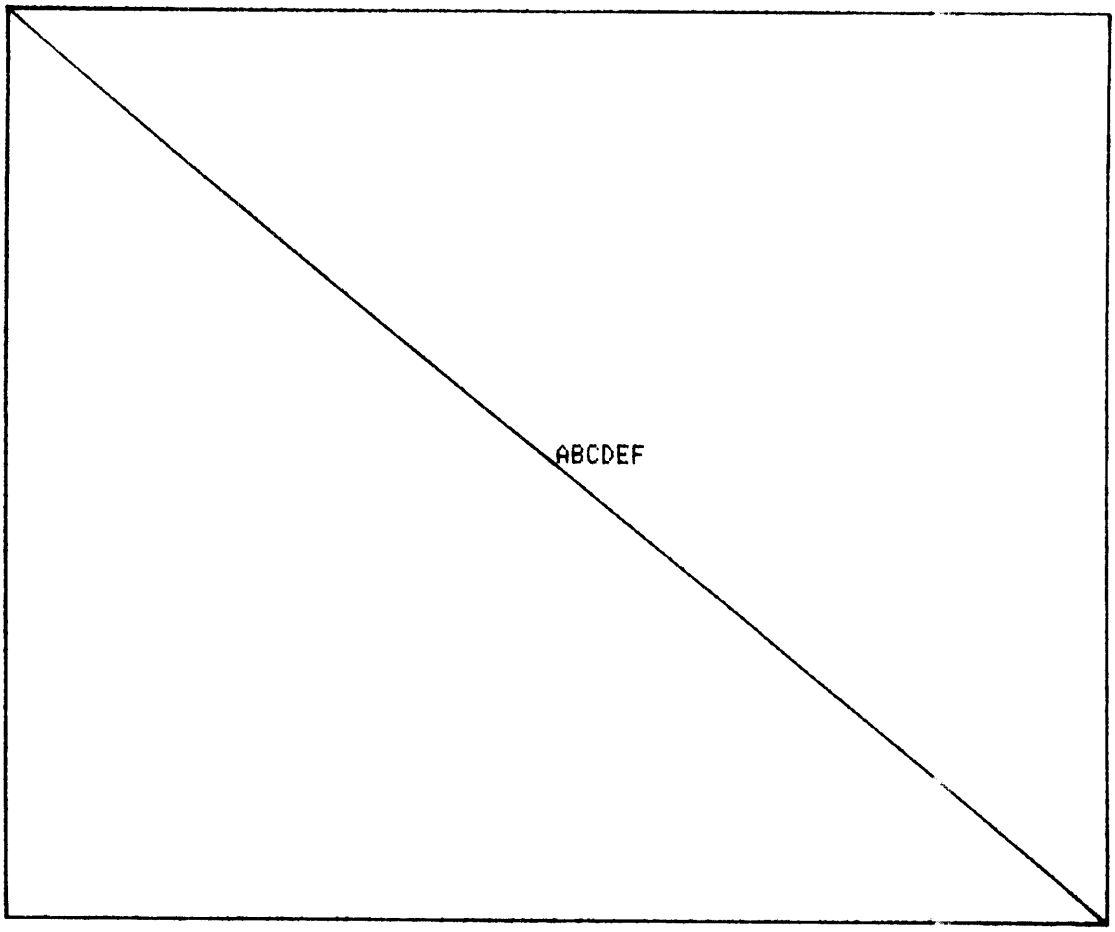
The beginning point of the second line is displaced by the characters printed by the PRINT command at statement 135. The DRAW command at statement 140 specifies only the end point to which the line will be drawn. It draws the line from wherever the graphic point is located when statement 140 is executed.

This example illustrates that care must be taken when lines and printed characters are output in the same program.

One way to avoid inadvertent distortion of graphic output is to keep track of the number of characters being output at a given PRINT statement. Then, by using the control characters described above, backspace and up-space to the beginning location of any character output. Subsequent graphic output may be continued as if nothing had been output at that point. The next example illustrates a way to do this. Press the PAGE key and enter the following statements:

```
DELETE ALL
100 PAGE
110 INIT
120 A$="ABCDEF"
130 MOVE 0,100
140 DRAW 65,50
150 PRINT A$;
160 FOR I=1 TO LEN(A$)
170 PRINT "H";
180 NEXT I
190 DRAW 130,0
200 HOME
210 END

RUN
```

Here is the resulting output:



Control H is a backspace character. The FOR . . . NEXT loop at statements 150 to 170 simply backspaces the number of characters output in A$. The DRAW at line 180 is then continued appropriately.

Statements 160 through 180 use only the number of characters in A$ to determine the graphic point's movement. This is appropriate when the PRINT command is placed in a subroutine. No positional information is then required by the subroutine. In the example program (above) statements 160 through 180 could easily be replaced by the following command: MOVE 65,50. This approach is simpler in the context of the example. However, it might not be appropriate when used in a different program. The example here simply shows an alternative way to perform the same function which requires different information. One way requires the number of characters in the character string which is printed. The other way requires the position on the display where it was printed. Both approaches are valid.

GRAPHIC INPUT

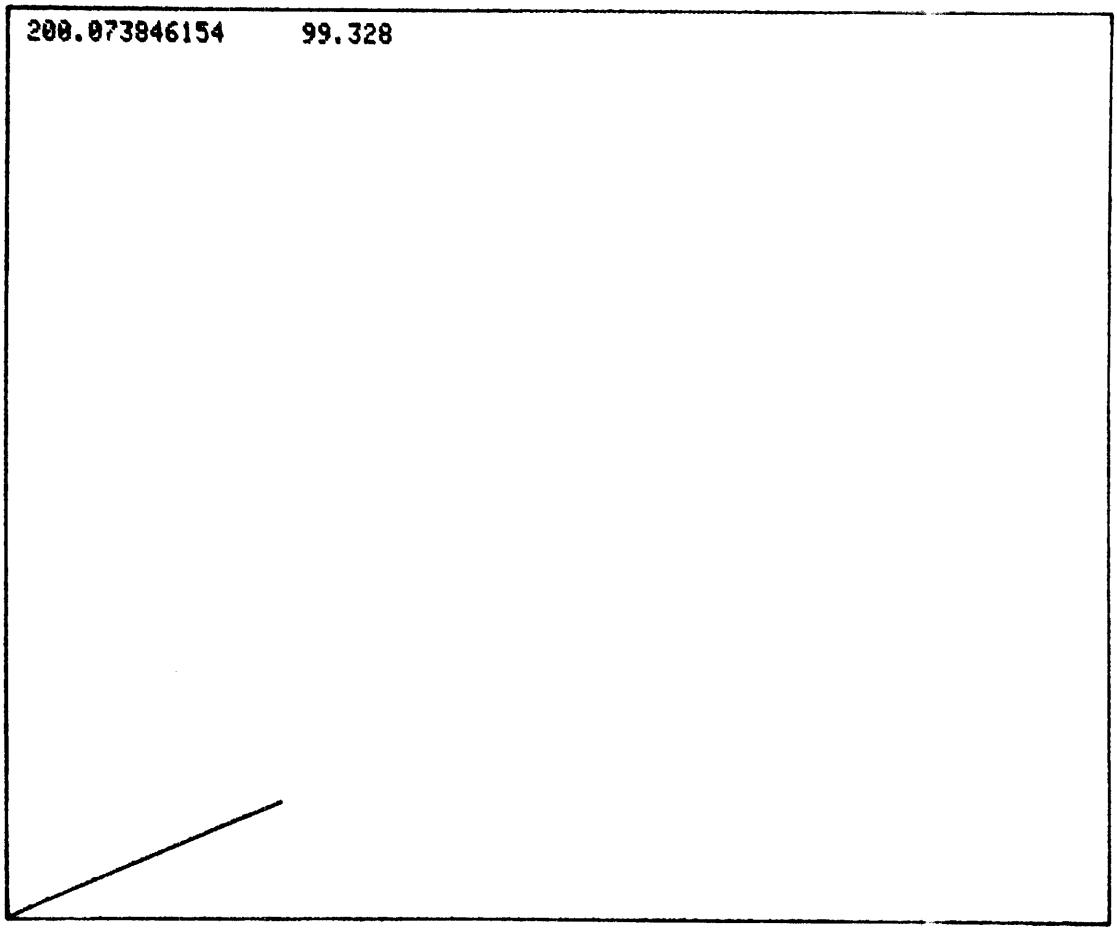
GIN

[Line number] GIN [I/O address] target variable for X coordinate

target variable for Y coordinate

Several commands in the Graphic System language are useful for finding out where the graphic point is located. The GIN or graphic input command is the principal means of performing this task. For example, press the PAGE key and enter the following statements:

```
DELETE ALL
100 PAGE
110 INIT
120 WINDOW 0,800,0,800
130 MOVE 0,0
140 DRAW 200,100
150 GIN A,B
160 HOME
170 PRINT A,B
180 END

RUN
```

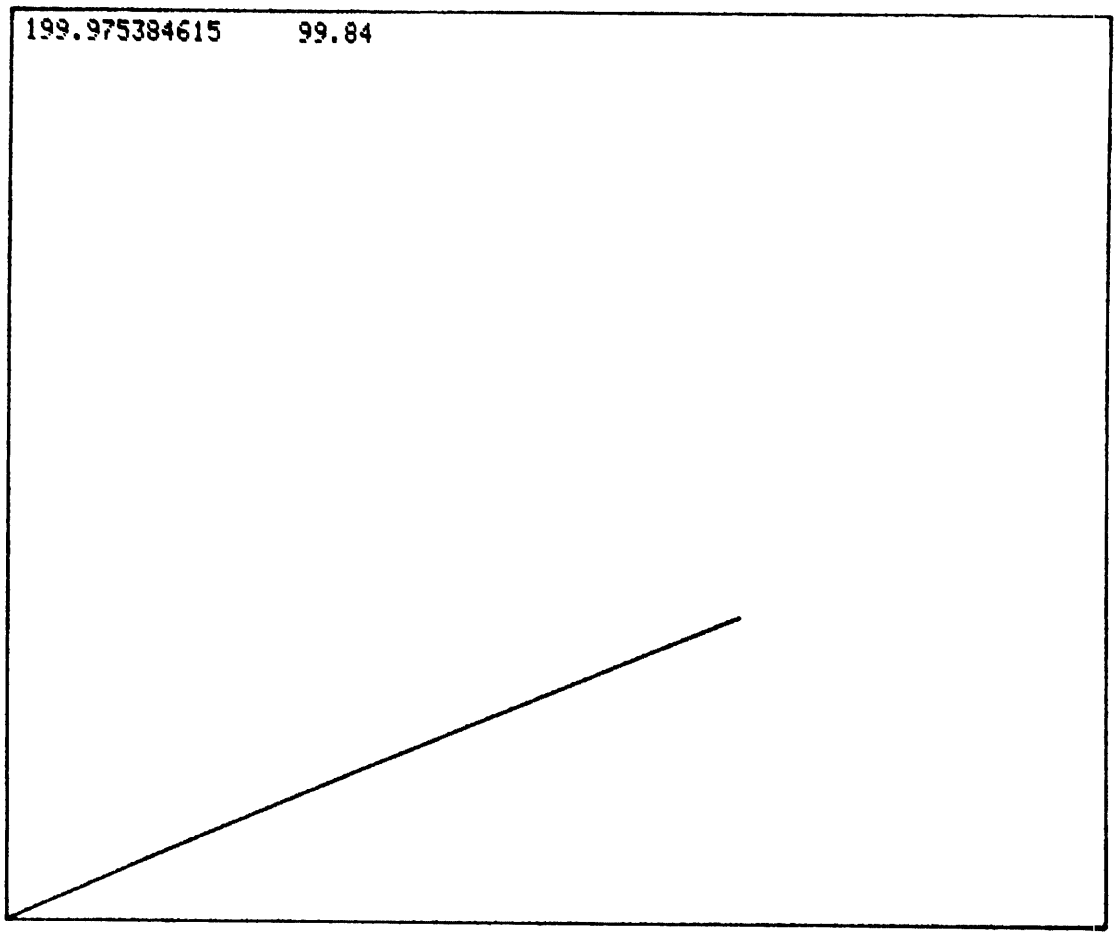


In the above example, the endpoint of the line shows the position of the graphic point when the GIN command is executed. The location of the graphic point in user data space determines what values are placed in the GIN command's target variables.

In a program such as the example above, it is unlikely that the values returned by the GIN command will exactly match the values in the DRAW command (statement 140). The accuracy of GIN is determined by the resolution of the graphic device being addressed, not on the GS. Each type of graphic device has a different resolution. For instance, the resolution of a CRT display, such as the one on the GS, is different from the resolution of a plotter. If a full size viewport is defined for the display, the values returned from GIN will have less than .125% error. This represents a worst-case accuracy (for a full size viewport) of better than 1 part in 800. In the above example, the window is defined to be 800 by 800 user units. An accuracy of better than 1 part in 800 means that the values returned by the GIN command will never differ more than 1 data unit from the values in the DRAW command.

The values returned by GIN are user data units. Running the above example with a different window illustrates the significance of this fact. Enter the following:

```
120 WINDOW 0,300,0,300
RUN
```



Because of the totally different window, the actual displayed location of the line's endpoint has changed. However, the values returned from GIN are virtually identical. The arguments of the DRAW command are unchanged. Therefore, the location of the line's endpoint in user data space is the same.

As in the previous example, the maximum error (in user data units) will be no greater than 1/800 of the defined window size.

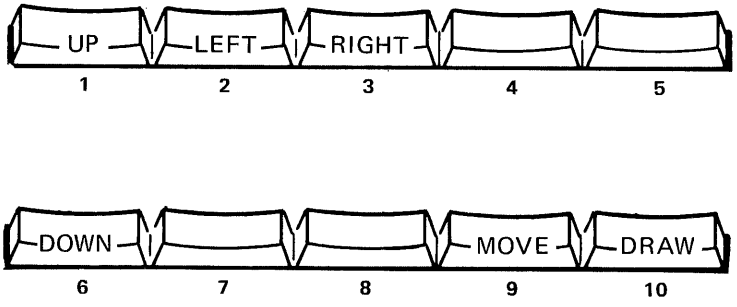
300*(1/800) = .375 user data units.

Below is an example which uses GIN. This program moves the position of the graphic point with the user definable keys. The position of the graphic point is always indicated by the flashing dot matrix which is normally seen on the display. Enter the following:

```
DELETE ALL
1 GO TO 100
4 RMOVE 0,1
5 RETURN
8 RMOVE -1,0
9 RETURN
12 RMOVE 1,0
13 RETURN
24 RMOVE 0,-1
25 RETURN
36 GIN X,Y
37 RETURN
40 GIN A,B
41 MOVE X,Y
42 DRAW A,B
43 X=A
44 Y=B
45 RETURN
100 INIT
110 PAGE
120 X=0
130 Y=0
140 MOVE 0,0
150 END

RUN
```

Here is a summary of the key functions and a diagram of each function's relative position:

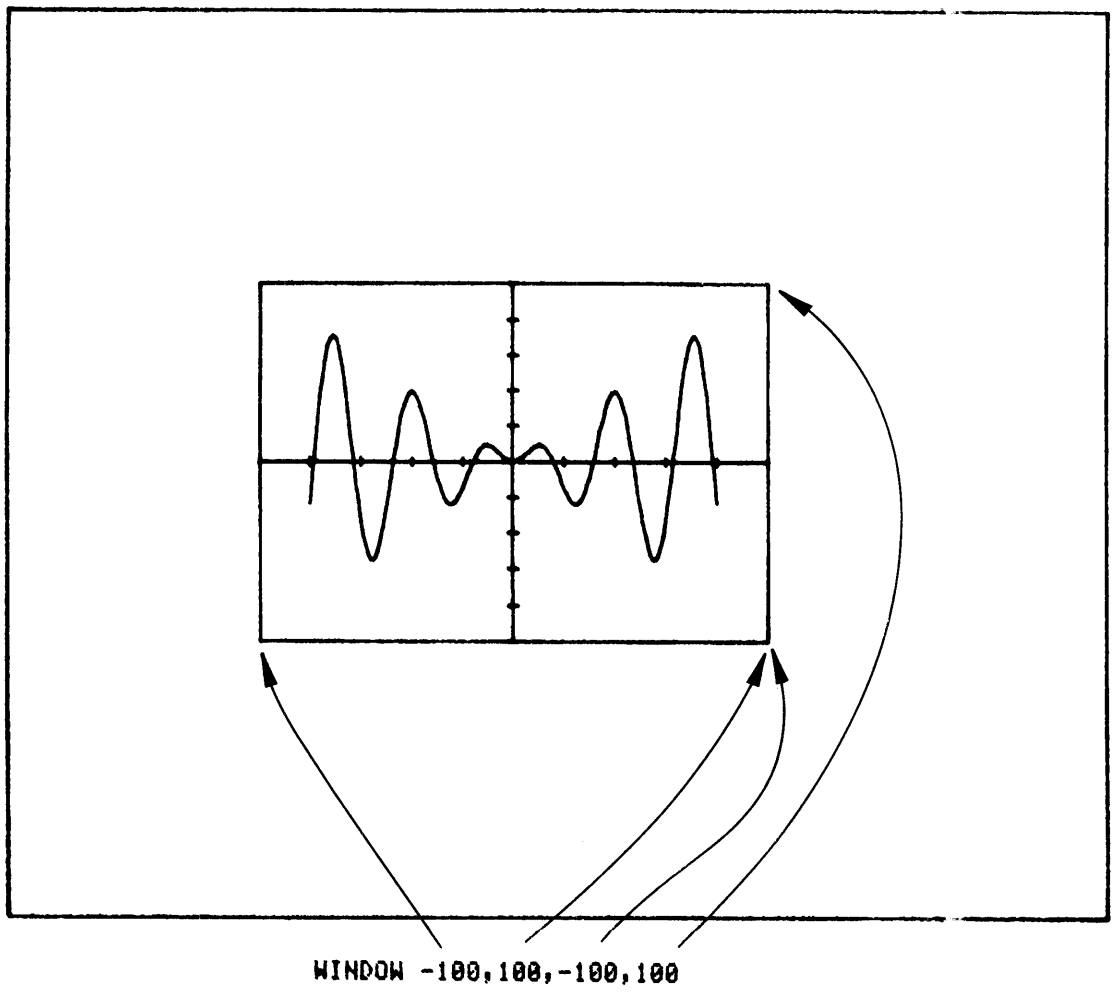
Key 1	Moves the graphic point up one data unit.
Key 2	Moves the graphic point to the left one data unit.
Key 3	Moves the graphic point to the right one data unit.
Key 6	Moves the graphic point down one data unit.
Key 9	Saves the current location of the graphic point as the beginning point of a line.
Key 10	Draws a line from the last specified beginning point to the current location of the graphic point.



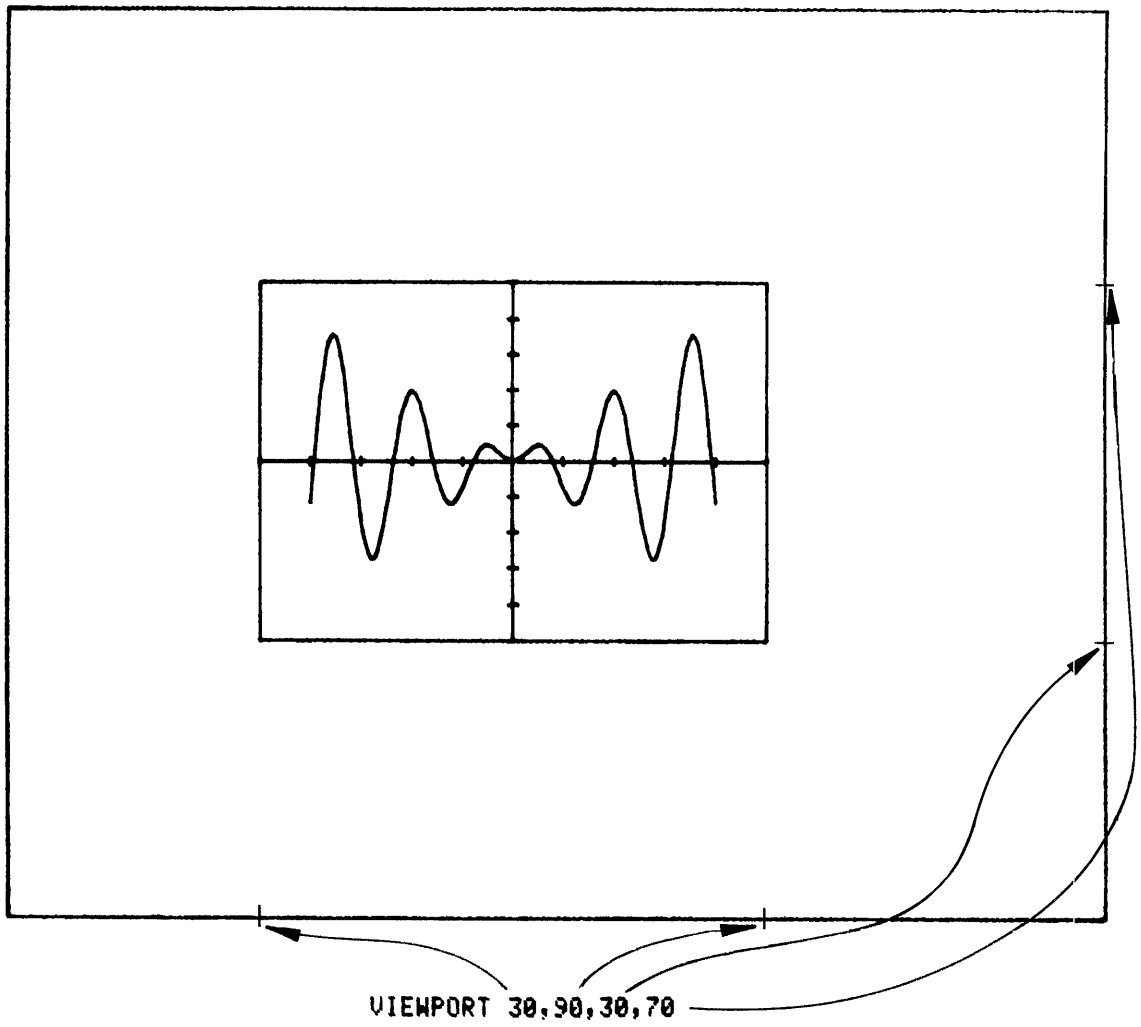
SUMMARY

The fundamental building block of drawn information is the line. Any line is defined by a beginning point and an end point. To draw lines on the Graphic System, the DRAW command is used. The arguments of the DRAW command determine the end point of a line. The starting point for output on the GS display is the graphic point. DRAW causes a line to be drawn from the current location of the graphic point to the location specified in the DRAW command's arguments. The MOVE command simply moves the position of the graphic point to the location specified by the MOVE command's arguments.

The WINDOW command specifies the minimum and maximum limits of displayed data values. These are called user data units. The arguments of MOVE and DRAW are interpreted to be expressed in these units.



The VIEWPORT command specifies the actual size and location of the rectangle on the display which contains data limits defined by WINDOW. The arguments of VIEWPORT are expressed in GRAPHIC DISPLAY UNITS. These units of measure are constant. GDU's are used to specify an actual physical location on the display.



WINDOW specifies what data is to be seen. VIEWPORT specifies where on the display it is located.

Both the WINDOW and VIEWPORT are set to default sizes whenever any of the following events occur:

- the GS is turned on
- any of these commands are executed:

```
INIT
OLD
DELETE ALL
```

WINDOW explicitly defines the data limits. WINDOW and VIEWPORT together imply scale factors. The SCALE command (with arguments expressed in user data units per GDU) explicitly defines horizontal and vertical scale factors. SCALE and VIEWPORT together implicitly define the data limits.

MOVE and DRAW specify a location in user data space which is "absolute", independent of the starting point. RMOVE and RDRAW specify a location relative to the present location of the graphic point.

ROTATE applies only to RMOVE and RDRAW. A positive argument produces counterclockwise rotation, a negative argument produces clockwise rotation.

If the arguments of a DRAW command are arrays, all the data points in the arrays are connected with lines. For example, if X and Y are arrays, the command DRAW X,Y draws a line to the point X(1),Y(1), then the point X(2),Y(2), and so forth. RDRAW, MOVE, and RMOVE also have this capability.

The AXIS command produces axis lines with tic marks if desired. If the intersection of the axis lines is not specified, a default location is computed.

The PRINT command is used to place character information on the display. The characters can be located using the MOVE command or any of several control characters. Since the graphic point determines the starting point for both lines and characters, care must be exercised when lines and characters are mixed in the same image.

The current location of the graphic point in user data space can be found with the GIN command. It returns values in user data units.

Section 2

DATA INPUT

ARRAYS

Any data to be graphed must first be input into the Graphic System. This section discusses input techniques appropriate for various types and sources of data. While each example program in this section can be run as listed, each one represents only the data input sub-section of a larger, more complete program.

In this section of the manual, the term "data value" denotes a single number while the term "data point" denotes a pair of data values used to specify a location in user data space.

Beginning at this point in the manual, the following assumption is made about all example programs: the GS memory has been cleared of any other program and data before the example is entered. Clearing memory is the function of the DELETE ALL command. It was used prior to each example in Section 1. There are a few instances where the example statements are to be added to a program already in memory. Each of these instances is specifically mentioned.

Data to be graphed may be stored in a group of simple (non-subscripted) variables. For example, the first value plotted can be stored in Scalar variable A, the second value in B and so forth. As can be quickly seen, editing, plotting and otherwise manipulating data stored in this fashion is very cumbersome. A better technique is the use of arrays. With data stored in arrays, it is easier to have the machine do more of the required work. Either one or two dimensional arrays can be appropriate for a given situation. Ways to use one dimensional arrays are discussed in most of this section. Changes necessary for the use of two dimensional arrays are discussed at the end of this section.

Before any data can be placed in an array, the array's size must be initialized so that it contains at least as many elements as the minimum number of data values expected. The DIM command is used for this purpose. Any program which uses the data in this array needs to know how many data values it contains. (This number may be less than the size of the array.) The user must set up some convention so that this may be accomplished easily.

The most versatile way to communicate how many data points the array contains is to dedicate one simple (non-subscripted) variable to contain the number of data values in the particular array. If any deletions or additions are made to the data array, then the number contained in the dedicated variable must also be decremented or incremented appropriately.

Another way of accounting for the number of data values in an array is to have the dedicated location (containing the number of data values in the array) be the first element in the array. This has several disadvantages. The index of a particular data value (in relation to the other data values) is not the same as its location in the array. For example, the sixth data value in a given array of data values is the seventh element in the array. Another major disadvantage has to do with retrieving the array after it has been placed onto an external storage device, such as tape. This will be discussed in more detail in the part of this section which deals with tape.

Another method is to write the program so that it will recognize some number, such as −999.999, as a non-valid data value. This number is then placed in every array element which does not contain a valid value. Along with the storage-related disadvantage of the previous method, this method does not allow rapid determination of the number of data values a given array contains.

All three of these methods require the same amount of storage: one more variable than the number of data values actually being stored. So the criteria of versatility and ease of use should determine which method is preferable. As described above, having a separate variable containing the number of data values is the best method for most applications.

FROM THE KEYBOARD

To enter data into the GS from the keyboard, the INPUT command is used. There are four fundamental methods for using the INPUT command to bring data into an array.

One way to place data from the keyboard into an array is with a FOR . . . NEXT loop, as in the following example:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N)
340 FOR I=1 TO N
350 INPUT A(I)
360 NEXT I
370 REM  END OF INPUT
380 END
```

First, the desired size of the array is placed into the variable N (statement 310). Statement 320 deletes the array. This is done to be sure that no other arrays are in memory with the same designation and that this array is dimensioned properly. Next, the size of array A is specified (statement 330). Then, a FOR . . . NEXT loop is used to cause the INPUT statement to be executed the appropriate number of times, with each data value inputted being placed into the correct array element.

There is a simpler way to perform the same task, taking advantage of the implied FOR . . . NEXT loop in the input statement. When the statement INPUT A is executed with variable A being an array, the GS automatically executes the INPUT statement enough times to fill the array. In effect, it sets up the FOR . . . NEXT loop of statements 340 to 350 in the previous example. So the last example can be replaced with the one below:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N)
340 INPUT A
350 REM  END OF INPUT
360 END
```

The input statement at line 340 will always start filling array A at its first element and continue until the array is filled. If either of these actions is inappropriate then a FOR . . . NEXT loop, such as the one shown in the previous example, must be set up. Using the intrinsic array capability of the INPUT command allows a series of input values to be separated by keys other than the carriage return. Some of them are the comman, the asterisk (*), the slash (/), or the up arrow (↑).

The other two methods of keyboard input (described below) involve setting up a continuous input loop which is then terminated in various ways. These methods share a common advantage (the exact number of elements in the array does not need to be specified in advance) and two common disadvantages: the array must be dimensioned to a size which is estimated to be greater than the largest number of data values expected to be input; and a check is required to ensure that the array size is not exceeded.

It is possible to set up a loop to input data points into an array so that a particular data value, specified in advance, terminates the loop. For example:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N)
340 A=0
350 C=0
360 INPUT K1
370 IF K1=-999.999 THEN 410
380 C=C+1
390 A(C)=K1
400 IF C<N THEN 360
410 REM   END OF INPUT
420 END
```

Again, the count of valid data values, C, is only incremented after the input of a valid number has been completed. When this method is used, one data value must be specified as the flag. This particular value cannot be a valid data value.

A way to minimize this disadvantage is to specify what the flag value is to be before the beginning of the input loop. For example:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
313 PRINT "VALUE OF TERMINATING FLAG = ";
317 INPUT T
320 DELETE A
330 DIM A(N)
340 A=0
350 C=0
360 INPUT K1
370 IF K1=T THEN 410
380 C=C+1
390 A(C)=K1
400 IF C<N THEN 360
410 REM  END OF INPUT
420 END
```

In this case, each data value input is compared with T, rather than a constant, to determine if the final data value has been entered.

Another method is to use the character string capabilities of the GS. If a string of letters is used to terminate the input loop, there is no requirement for any numeric value to serve as a flag. This method uses the string function VAL (which converts a character string composed of the characters 0.1, . . . ,9,E+,—) to a number.

For example:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N),K$(20)
340 A=0
350 C=0
360 INPUT K$
370 IF K$="DONE" THEN 410
380 C=C+1
390 A(C)=VAL(K$)
400 IF C<N THEN 360
410 REM  END OF INPUT
420 END
```

Using the program example above, data values can be entered into successive elements of array A until the character string DONE is entered. This will cause the GS to exit from the loop and begin executing the statements following the loop. Memory space will be conserved if the string variable K$ is dimensioned to the length of the longest anticipated terminating character string. The default size for K$ is 72 characters, an unneeded length in this context. A statement such as 330 DIM A(N),K$(20) would be appropriate for this example, since most of numeric data values require fewer than 20 characters to express.

It is also possible for the terminating character string to be specified just prior to the input loop. For example:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 PRINT "TERMINATING STRING = ";
330 DIM T$(20)
340 INPUT T$
350 DELETE A
360 DIM A(N),K$(20)
370 A=0
380 C=0
390 INPUT K$
400 IF K$=T$ THEN 440
410 C=C+1
420 A(C)=VAL(K$)
430 IF C<N THEN 390
440 REM END OF INPUT
450 END
```

Both T$ and K$ have been dimensioned to conserve memory space. If not dimensioned, each would have space for 72 characters. When a character string identical to the one entered into T$ is input, the test in statement 400 is true and the loop is exited via statement 430.

There is one terminating string which is very convenient to use: the null string. This is the character string which contains nothing. In the previous example, T$ contains the null string if the user enters only a carriage return in response to the "TERMINATING STRING = " request. Another way to specifying the null string as the terminating string is shown in the following example.

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 PRINT "TERMINATING STRING = ";
330 DIM T$(20)
340 INPUT T$
350 DELETE A
360 DIM A(N),K$(20)
370 A=0
380 C=0
390 INPUT K$
400 IF K$=T$ THEN 440
410 C=C+1
420 A(C)=VAL(K$)
430 IF C<N THEN 390
440 REM   END OF INPUT
450 END
```

To terminate this loop, merely follow the carriage return which enters the last data value with an extra carriage return. The extra carriage return inputs a null string, which is recognized and causes the loop to be exited.

Another technique for terminating an input loop is to set up a user definable key as the terminating event. For example:

```
1 GO TO 290
4 GO TO 390
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N)
340 A=0
350 SET KEY
360 FOR I=1 TO N
370 INPUT A(I)
380 NEXT I
390 REM  END OF INPUT
400 END
```

In the previous example, pressing the user definable key 1 causes an immediate branch to statement 390. When statement 390 is executed, I will be one greater than the number of data values input.

A similar example:

```
1 GO TO 290
4 GO TO 380
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N)
340 A=-999.999
350 SET KEY
360 INPUT A
370 REM  END OF INPUT
380 FOR I=1 TO N
390 IF A(I)=-999.999 THEN 410
400 NEXT I
410 C=I-1
420 REM  C IS NUMBER OF VALID DATA VALUES
430 END
```

This example uses the array capability of the GS to do much of the required work. Statements 380 through 410 are required because, with this approach, there is no intrinsic way to determine how many values have been input into the array. (In the previous example, the value of I after leaving the FOR . . . NEXT loop provided that information.) Statement 340 is needed to initialize all the array elements to a specified dummy value. Any element containing this value has not had a valid data value placed into it. Statements 380 through 410 determine how many valid data values have been entered.

Prior to graphing the input data, the minimum and maximum must be known. There are two times this can be done: as the data is entered, and just prior to graphing. If variable R1 is the minimum and R2 is the maximum, the following is an example of how the minimum and maximum might be determined at entry time:

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N),K$(20)
340 A=0
343 R1=1.0E+300
347 R2=-1.0E+300
350 C=0
360 INPUT K$
370 IF K$="" THEN 410
380 C=C+1
390 A(C)=VAL(K$)
393 R1=R1 MIN A(C)
397 R2=R2 MAX A(C)
400 IF C<N THEN 360
410 REM  END OF INPUT
420 END
```

The min-max determination adds only four statements (343, 347, 393, 397). The two initialization statements (343 and 347) set the minimum to an artifically large positive number and the maximum to an artificially large negative number. This ensures that any given data entry will be detected as a minimum or a maximum. (If statements 343,347,393 and 397 are removed, the above example is identical to the one on page 2-6 which uses the null string to terminate the input loop.)

The other time to determine the minimum and maximum is just prior to graphing. This technique has the advantage of including any changes to the data which occurred after it was entered. In the following example, C is the number of data items in array A and R1 and R2 are the minimum and maximum respectively.

```
290 PAGE
300 PRINT "SIZE OF ARRAY = ";
310 INPUT N
320 DELETE A
330 DIM A(N),K$(20)
340 A=0
350 C=0
360 INPUT K$
370 IF K$="" THEN 410
380 C=C+1
390 A(C)=VAL(K$)
400 IF C<N THEN 360
410 REM  END OF INPUT, FIND MIN & MAX
420 R1=1.0E+300
430 R2=-1.0E+300
440 FOR I=1 TO C
450 R1=R1 MIN A(I)
460 R2=R2 MAX A(I)
470 NEXT I
480 END
```

This technique of scanning after entry is useful when data is entered from sources other than the keyboard, as examples later in this section demonstrate.

Use of the minimum and maximum values is discussed in Section 3 of this manual.

EDITING AN ARRAY

Editing a data array already in the GS memory is sometimes necessary. Data editing is made up of five processes:

- Listing current data
- Replacing a data value with a new one
- Deleting a data value
- Appending a data value to the end of the current array
- Inserting a data value into the current array.

Referring to any data item within an array can only be done two ways: by referring to the item's position or index within the array (e.g., A(4)), or by referring to the value of the data item (e.g., the first data item equal to 3). Because a data item's index is so often used for reference, and because the five editing processes each have a different effect on the contents of a data array, care must be exercised during an editing operation.

The five editing processes are now discussed in the order mentioned, with examples. The array used, A, can contain no more than N data values. In the following five examples, N is 20. C is the working size of the array. It represents the number of valid data values currently contained in array A. In each of the following examples, 10 is the beginning value of C. Each of the first ten elements of A is filled with a value which is ten times the element's index. (Statements 160 through 180 in each program perform this function).

Listing Data

When listing an array, it is very useful to print each item beside that item's index. This simplifies further reference to each data point. Here is a way this can be done:

```
100 PAGE
110 INIT
120 N=20
130 C=10
140 DIM A(N)
150 A=0
160 FOR I=1 TO C
170 A(I)=10*I
180 NEXT I
190 REM  LIST VALUES IN ARRAY A
200 IF C>0 THEN 230
210 PRINT "NO DATA VALUES IN ARRAY A"
220 GO TO 260
230 FOR I=1 TO C
240 PRINT I,A(I)
250 NEXT I
260 END
```

These statements cause each data item to be printed on the display just to the right of that data item's index. There is a check to ensure that the program will not attempt to print an empty array (statement 200). If C is zero, there are no valid data values in the array. An array with no data values should not be listed.

In the next four example programs the index number of an array element is required before the specific editing function can be performed. If the null string is entered when the program requests the element index (by pressing only the carriage return key at that point), program execution is terminated in each case.

Changing Data

If a given data value is to be changed, the following statements can be used; they allow changes to be made very easily.

```
100 PAGE
110 INIT
120 N=20
130 C=10
140 DIM A(N)
150 A=0
160 FOR I=1 TO C
170 A(I)=10*I
180 NEXT I
190 REM  CHANGE DATA VALUE
200 PRINT "INDEX OF ITEM TO CHANGED = ";
210 INPUT I$
220 IF I$="" THEN 310
230 I=VAL(I$)
240 IF I=>1 AND I<=C THEN 270
250 PRINT "INDEX IS OUT OF RANGE"
260 GO TO 310
270 PRINT "CURRENT VALUE OF A(";I;") IS  ";A(I)
280 PRINT "CHANGE THIS TO  ";
290 INPUT A(I)
300 GO TO 190
310 END
```

Included is a check for a proper item number (statement 240). The current item's value is printed prior to the change being made. In this example, the input to the process is treated as a string which is tested to see if it is the terminating string. If it is, the process is exited (in the example, control branches to statement 310). If it is not the terminating string, then the change process is continued.

Neither of the previous two examples resulted in any change of the number of data items in the array. All three following processes will result in a change in the number of data items in the array.

Appending Data

If a given data point is to be appended to an array, the following statements will accomplish the task:

```
100 PAGE
110 INIT
120 N=20
130 C=10
140 DIM A(N)
150 A=0
160 FOR I=1 TO C
170 A(I)=10*I
180 NEXT I
190 REM  APPEND DATA VALUE
200 IF C<N THEN 230
210 PRINT "ARRAY IS FILLED"
220 GO TO 290
230 PRINT "VALUE OF ITEM TO BE APPENDED = ";
240 INPUT V$
250 IF V$="" THEN 290
260 C=C+1
270 A(C)=VAL(V$)
280 GO TO 200
290 END
```

The number of data points in the array (C) is changed only after valid input has been entered. The array A has been dimensioned to contain N data elements. A check has been included so that the array's size will not be exceeded and thus cause an error (check is statement 200).

Deleting Data

If a given data point is to be deleted from an array, the following statements can be used:

```
100 PAGE
110 INIT
120 N=20
130 C=10
140 DIM A(N)
150 A=0
160 FOR I=1 TO C
170 A(I)=10*I
180 NEXT I
190 REM  DELETE DATA VALUE
200 IF C>0 THEN 230
210 PRINT "ARRAY IS EMPTY"
220 GO TO 370
230 PRINT "INDEX OF ITEM TO BE DELETED = ";
240 INPUT I$
250 IF I$="" THEN 370
260 I=VAL(I$)
270 IF I=>1 AND I<=C THEN 300
280 PRINT "INDEX IS OUT OF RANGE"
290 GO TO 200
300 PRINT "VALUE DELETED = ";
310 PRINT A(I)
320 FOR J=I+1 TO C
330 A(J-1)=A(J)
340 NEXT J
350 C=C-1
360 GO TO 200
370 END
```

The program must "ripple" through the array, moving all data items after the deleted one toward the beginning of the array by one position.

Inserting Data

The last kind of editing mentioned is the insert process, allowing a data value to be inserted at any point in the array. This process must also "ripple" through the array, but it moves all the data items toward the end of the array by one position.

```
100 PAGE
110 INIT
120 N=20
130 C=10
140 DIM A(N)
150 A=0
160 FOR I=1 TO C
170 A(I)=10*I
180 NEXT I
190 REM  INSERT DATA VALUE
200 IF C<N THEN 230
210 PRINT "ARRAY IS FILLED"
220 GO TO 370
230 PRINT "INDEX OF ITEM TO BE PRECEDED = ";
240 INPUT I$
250 IF I$="" THEN 370
260 I=VAL(I$)
270 IF I=>1 AND I<=C THEN 300
280 PRINT "INDEX IS OUT OF RANGE"
290 GO TO 200
300 FOR J=C TO I STEP -1
310 A(J+1)=A(J)
320 NEXT J
330 C=C+1
340 PRINT "VALUE TO BE INSERTED = ";
350 INPUT A(I)
360 GO TO 190
370 END
```

The working size of the array (C) must be incremented to make room for the insertion (statement 330). Statement 270 is a check to ensure that the item preceded is within the working size of the array. Statement 200 is a check to ensure that the array is large enough to contain the insertion.

FROM A FUNCTION

Graphing a single-valued function is fundamentally similar to graphing an array of data values. However, there are some differences in the way the data is supplied to the graphing program.

The most satisfactory way to set up the graphing of a function is to place the function to be graphed within a FOR . . . NEXT loop. This can be done directly, as in the first program example in Introduction. It can also be done indirectly, by placing within the FOR. . .NEXT loop a GOSUB reference to the function. For example:

```
    .
    .
    .
100 FOR X=B TO E STEP I
110 GOSUB 6000
120 DRAW X,Y
130 NEXT X
    .
    .
    .
6000 REM  SAMPLE FUNCTION
6010 Y=SIN(X)
6020 RETURN
```

Any function to be graphed need only be keyed into the GS starting at line 6000 and followed by a RETURN statement.

Notice that the FOR . . . NEXT loop requires certain information about the function's independent variable (called X in this example). Three values must be specified: the beginning value (called B in this example), the ending value (called E in this example), and the increment (called I in this example). This data allows the GS to successively evaluate the function within a specified domain at specified intervals.

A minor change is necessary if graphing a specified number of points within the domain of the function is desired. The FOR . . . NEXT loop lends itself to using a specified increment. In order to graph a specified number of points, divide the total domain (the maximum minus the minimum) by the number of points desired. The quotient will be the increment. With this approach, statement 100 of the previous example would look like the following:

```
100  FOR X = B TO E STEP (E—B)/N
```

In this case, N is the desired number of points to be graphed.

In order to graph the function, two additional data are required: the minimum and maximum of the function within the specified domain. Unless these values are estimated, they must be determined by evaluating the function throughout its domain just as if it is being graphed. An example of how this is done follows (this example cannot be run as listed):

```
      .
      .
      .
100 REM  FIND MINIMUM AND MAXIMUM
110 REM  MINIMUM IS M1, MAXIMUM IS M2
120 M1=1.0E+300
130 M2=-1.0E+300
140 FOR X=B TO E STEP I
150 GOSUB 6000
160 M1=M1 MIN Y
170 M2=M2 MAX Y
180 NEXT X
190 WINDOW B,E,M1,M2

      .
      .
      .
6000 REM  SAMPLE FUNCTION
6010 Y=SIN(X)
6020 RETURN
```

The minimum and maximum resulting from this calculation can be used directly in a WINDOW statement. This ensures that the viewport is filled with the appropriate part of the user data space.

Parametric functions are handled in much the same way as simple functions. (This manual calls Y = SIN(X) a simple function and

```
Y = SIN(T)
X = COS(T)
```

a parametric function.)

To graph a parametric function, or to find its minimum and maximum, the only change needed in the above examples is in the variables in the FOR and NEXT statements (this example cannot be run as listed):

```
    .
    .
    .
100 FOR T=B TO E STEP I
110 GOSUB 6000
120 DRAW X,Y
130 NEXT T
    .
    .
    .
6000 REM  SAMPLE PARAMETRIC FUNCTION
6010 Y=SIN(T)
6020 X=COS(T)
6030 RETURN
```

It is possible to have the same program graph both simple and parametric functions. A convention must be made as to which variables are dependent and independent in simple and parametric functions. The chart shows the convention used in the following example:

	Simple Function	Parametric Function
Independent Variable	X	T
Dependent Variable	Y	X & Y

```
    .
    .
    .
100 FOR T=B TO E STEP I
110 X=T
120 GOSUB 6000
130 DRAW X,Y
140 NEXT T
    .
    .
    .
6000 REM  SIMPLE FUNCTION
6010 Y=SIN(X)
6020 RETURN
```

```
    .
    .
    .
100 FOR T=B TO E STEP I
110 X=T
120 GOSUB 6000
130 DRAW X,Y
140 NEXT T
    .
    .
    .
6000 REM  PARAMETRIC FUNCTION
6010 X=COS(T)
6020 Y=SIN(T)
6030 RETURN
```

The function at line 6000 will be evaluated properly regardless of whether it is simple or parametric.

FROM TAPE

Any discussion of graphing data stored on tape must be prefaced with a discussion of appropriate ways to store data onto tape. The GS internal tape drive is a sequential access device. This means that in order to read a data item at the end of a tape file, the whole file must be read in first. Reading a data item that precedes one which was just read is a two step process. First, the tape must be positioned back to the beginning of the file. Then, in order to find the desired item, the file must be re-read up to the desired item's location. (The tape can only be positioned within a file by reading data.) This means that the kind of data manipulation done by a program should influence the way it stores and reads data on the tape. Here is one method of storing data on the tape which is suitable for data to be graphed. The first value to be stored in a tape file is the number of graphed points which will fill the remainder of the file. Each graphed point might typically require 1, 2, or 3 numeric values to describe. If, for example, the data stored on the tape is to be used in a two-variable (or X—Y) graph, each point on the graph will require two data values to describe it: an X value (usually used for the graphed point's horizontal location) and a Y value (usually used for the vertical location). Therefore N, the first value in the tape file, will be equal to the number of pairs of data items in that tape file. The total number of data items in that file will be (2*N)+1. The best way to store these items is with the X and Y data values for a given graphed point adjacent to each other on the tape. For a two-variable graph's data, the tape file looks like this:

N,X(1),Y(1),X(2),Y(2), . . . X(N—1),Y(N—1),X(N),Y(N)

Below is a program which will write and read files with this arrangement:

```
100 PAGE
110 INIT
120 REM WRITE THE DATA FILE
130 C=20
140 FIND 0
150 MARK 1,10*(2*C+1)
160 FIND 1
170 WRITE C
180 FOR I=1 TO C
190 WRITE 10*I,10*0.1
200 NEXT I
210 CLOSE
220 REM READ THE DATA FILE
230 FIND 1
240 READ @33:N
250 FOR I=1 TO N
260 READ @33:X,Y
270 PRINT X,Y
280 NEXT I
290 END
```

Prior to writing data to tape, the tape must be marked into files. The file being written to must be MARKed large enough to contain the data to be stored into it. Each data value stored on the tape with the WRITE command requires 10 bytes. Therefore, the number of bytes required for a given file is 10 times the number of data values to be stored. The storage arrangement described above includes an extra value containing the number of data points in the file. The number of data values stored in the file is the product of the number of graphed points and the number of data values per point, plus one. The "plus one" reserves space for the extra data value (at the beginning of the file) which contains the number of data points within the file. If C is the number of graphed points and P is the number of data values per point, then the number of bytes of space in a tape file required to contain this data is 10*(P*C+1). Statement 150 performs this function in the above example.

Statement 190 writes sample data onto the tape in the order described. The CLOSE command (statement 210) should be executed after the last data item has been written to a file. It ensures that the data has been properly written to the tape.

To read the tenth data point on the tape, the following program can be used:

```
100 PAGE
110 INIT
120 REM WRITE THE DATA FILE (C IS THE NUMBER OF DATA PAIRS)
130 C=20
140 FIND 0
150 MARK 1,10*(2*C+1)
160 FIND 1
170 WRITE C
180 FOR I=1 TO C
190 WRITE 10*I,10*0.1
200 NEXT I
210 CLOSE
220 REM P IS THE INDEX OF THE DATA PAIR TO BE READ
230 P=10
240 FIND 1
250 READ @33:N
260 IF P<=N THEN 340
270 PRINT "INDEX IS GREATER THAN NUMBER OF PAIRS IN FILE"
280 GO TO 340
290 FOR I=1 TO P-1
300 READ @33:Q,Q
310 NEXT I
320 READ @33:X,Y
330 PRINT "10TH DATA PAIR IS ";X,Y
340 END
```

The Q variables in statement 300 are used merely to allow the READ statement to move the tape; Q is a "dummy" variable having no direct usefulness. Use of the variable P (in statements 230, 260, and 290) is not necessary. The constant 10 could have been used in this case. The variable P was used to show how any data point within the file may be accessed with this method.

The method of storing data on tape described in this section has one inherent disadvantage. It requires the user to keep track of how many data values are associated with each data point. There is no intrinsic indication on the tape whether each data point requires one, two, or more data values to represent it. It is imperative to remember whether there are one, two, or more data values for each data point within a file.

An easier way to store data on the tape is to use the implied looping of statements which have arrays for arguments. For example:

```
100 PAGE
110 INIT
120 REM WRITE THE DATA FILE (C IS THE NUMBER OF PAIRS)
130 C=20
140 DIM X(C),Y(C)
150 FIND 0
160 MARK 1,10*(2*C+1)
170 FIND 1
180 FOR I=1 TO C
190 X(I)=10*I
200 Y(I)=10*I+0.1
210 NEXT I
220 WRITE C,X,Y
230 CLOSE
240 INIT
250 FIND 1
260 READ @33:N
270 DIM A(N),B(N)
280 READ @33:A,B
290 PRINT A,B
300 END
```

The data is stored on the tape in the following arrangement:

$$N,X(1),X(2),\ldots X(N-1),X(N),Y(1),Y(2),\ldots Y(N-1,Y(N)$$

All of the data is written onto the tape by statement 220 (above). All of the data is read from the tape by statements 260 and 280.

Although this method is very simple and easy to use, it has a significant disadvantage. It requires a GS whose memory is large enough to hold all the data internally at one time. Since the X and Y data values for a given point are located at two different places on the tape, at least one FOR . . . NEXT loop is required to access one data point.

```
100 PAGE
110 INIT
120 REM WRITE THE DATA FILE (C IS THE NUMBER OF PAIRS)
130 C=20
140 DIM X(C),Y(C)
150 FIND 0
160 MARK 1,10*(2*C+1)
170 FIND 1
180 FOR I=1 TO C
190 X(I)=10*I
200 Y(I)=10*I+0.1
210 NEXT I
220 WRITE C,X,Y
230 CLOSE
240 INIT
250 FIND 1
260 REM P IS THE INDEX OF THE DATA PAIR TO BE READ
270 P=10
280 READ @33:N
290 DIM A(N)
300 READ @33:A
310 FOR I=1 TO P-1
320 READ @33:Q
330 NEXT I
340 READ @33:B1
350 PRINT "10TH DATA PAIR IS ";A(P),B1
360 END
```

This kind of program is necessary unless the entire contents of both X and Y arrays have been read into memory.

The first method described, with the X and Y data values for a given point adjacent to each other on the tape, is the superior method in cases where all data to be graphed cannot be contained in the GS memory at one time. If all the data can be contained in the GS memory at one time, then the method shown immediately above is preferable.

Finding the minimum and maximum of the data prior to graphing is necessary when the data comes from tape. This can be done two ways: if all the data is contained in memory at one time, the min-max scanner on page 2-9 can be used; if all the data is not in memory, the tape can be scanned using the following kind of program (X and Y are the two data values for each data point, X1 and Y1 are the minimums, and X2 and Y2 are the maximums).

```
100 PAGE
110 INIT
120 REM WRITE THE DATA FILE (C IS THE NUMBER OF DATA PAIRS)
130 C=20
140 FIND 0
150 MARK 1,10*(2*C+1)
160 FIND 1
170 WRITE C
180 FOR I=1 TO C
190 WRITE 10*I,10*I+0.1
200 NEXT I
210 CLOSE
220 REM FIND MIN AND MAX OF DATA ON TAPE
230 X1=1.0E+300
240 Y1=1.0E+300
250 X2=-1.0E+300
260 Y2=-1.0E+300
270 FIND 1
280 READ @33:N
290 FOR I=1 TO N
300 READ @33:X,Y
310 X1=X1 MIN X
320 X2=X2 MAX X
330 Y1=Y1 MIN Y
340 Y2=Y2 MAX Y
350 NEXT I
360 PRINT " ","MINIMUM","MAXIMUM"
370 PRINT "X",X1,X2
380 PRINT "Y",Y1,Y2
390 END
```

After the FOR . . . NEXT loop at statements 290 through 350 is complete, X1 will contain the minimum X data value, X2 will contain the maximum X data value, Y1 will contain the minimum Y data value and Y2 will contain the maximum Y data value. No arrays or other large areas of memory were reserved or required. These minimum and maximum values can be used directly in a WINDOW command, allowing the data on tape to be graphed one pair of values at a time. The technique above is important because it is a key to graphing more data than the GS can contain in memory at one time.

Summary

The procedure described above is one approach to storing data arrays on tape using READ and WRITE. Here is a summary of that approach.

If the GS memory is large enough to contain all the data of interest, then the most convenient way to store and retrieve data on tape is to use the inherent looping of the WRITE and READ commands. (In the examples below, F is the number of the file to contain the data and N is the number of data points. Each data point can be specified with one, two, or three data values. Data is contained in arrays A, B, or C, and in simple variables X, Y, or Z. None of the program segments below can be run as listed.)

For one data value per point:

To write data:

```
      .
      .
      .
200 DIM A(N)
210 FIND F
220 MARK 1,10*(N+1)
230 FIND F
240 WRITE N,A
250 CLOSE
```

The tape file looks like:

N,A(1),A(2), . . . A(N−1),A(N)

To read data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 READ @33:N
220 DELETE A
230 DIM A(N)
240 READ @33:A
```

For two data values per point:

To write data:

```
      .
      .
      .
      .
      .
      .
200 DIM A(N),B(N)
210 FIND F
220 MARK 1,10*(2*N+1)
230 FIND F
240 WRITE N,A,B
250 CLOSE
```

The tape file looks like:

N,A(1),A(2), . . . A(N),B(1),B(2), . . . B(N)

To read data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 READ @33:N
220 DELETE A,B
230 DIM A(N),B(N)
240 READ @33:A,B
```

For three data values per point:

To write data:

```
      .
      .
      .
      .
      .
      .
200 DIM A(N),B(N),C(N)
210 FIND F
220 MARK 1,10*(3*N+1)
230 FIND F
240 WRITE N,A,B,C
250 CLOSE
```

The tape file looks like:

N,A(1), . . . ,A(N),B(1), . . . ,B(N),C(1), . . . ,C(N)

To read data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 READ @33:N
220 DELETE A,B,C
230 DIM A(N),B(N),C(N)
240 READ @33:A,B,C
```

When data with more than one value per point is stored on tape, an additional consideration becomes important. When using the approach in the above six examples, the data values for a given point are not placed on the tape in adjacent locations. Retrieving the data values for just one point is cumbersome unless the GS memory is large enough to contain all data values at one time.

If all the data cannot be contained within the GS memory at one time, then the above approach is not suitable. Below is another method in which data values are stored and retrieved individually.

For one data value per point:

To write data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 MARK 1,10*(N+1)
220 FIND F
230 WRITE N
240 FOR I=1 TO N
250 WRITE X
260 NEXT I
270 CLOSE
```

The tape file looks like:

$$N,X(1),X(2),\ldots X(N-1),X(N)$$

To read data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 READ @33;N
220 FOR I=1 TO N
230 READ @33:X
240 NEXT I

      .
      .
      .
      .
      .
```

For two data values per point:

To write data:

```
      .
200 FIND F
210 MARK 1,10*(2*N+1)
220 FIND F
230 WRITE N
240 FOR I=1 TO N
250 WRITE X,Y
260 NEXT I
270 CLOSE
```

The tape file looks like:

N,X(1), Y(1),X(2),Y(2), . . . X(N−1),Y(N−1),X(N),Y(N)

To read data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 READ @33:N
220 FOR I=1 TO N
230 READ @33:X,Y
240 NEXT I
```

For three data values per point:

To write data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 MARK 1,10*(3*N+1)
220 FIND F
230 WRITE N
240 FOR I=1 TO N
250 WRITE X,Y,Z
260 NEXT I
270 CLOSE
```

The tape file looks like:

N,X(1),Y(1),Z(1),X(2),Y(2),Z(2), . . . , X(N),Y(N),Z(N)

To read data:

```
      .
      .
      .
      .
      .
      .
200 FIND F
210 READ @33:N
220 FOR I=1 TO N
230 READ @33:X,Y,Z
240 NEXT I
      .
      .
      .
MAG TAPE ERROR IN LINE 94 - MESSAGE NUMBER 55
```

The WRITE and READ @33: commands used in this section pertain to BINARY data files. Numbers in binary data files are in the same form as they are internally in the GS. Data can also be stored and retrieved on tape with the following commands: PRINT @33: and INPUT @33: . (The primary address of 33 refers to the internal tape drive on the GS.) These commands pertain to ASCII data files. Since numbers in ASCII data files are strings of ASCII characters, ASCII data files can be read by devices outside the GS family. Binary data files can be read only by another GS. However, data stored in ASCII data files takes longer for the GS to transfer to and from tape because an additional conversion process must be performed. All the example programs in this section using the WRITE and READ @33: commands would remain virtually unchanged if the PRINT @33: and INPUT @33: commands were substituted. Only the amount of space reserved on the tape for each data value would need changing. In a binary file, 10 bytes must be MARKed. In an ASCII file, 18 bytes must be MARKed for each data value. In virtually all instances, it is best to use the WRITE and READ @33: commands to store and retrieve data on tape.

Section 3

GRAPHING

INITIAL CONSIDERATIONS

Before any graphing can be done, the minimum and maximum of the data or function to be graphed must be established. This can be done by scanning the actual data values, evaluating the function over the specified domain of interest, or by merely estimating. In any event, this must be done to provide values to use in the WINDOW statement, or to otherwise specify what will be graphed.

The next consideration is how the data or function will be displayed. For graphing one data array, a single valued function or other data where there is some logical connection between adjacent points, a line drawn through all the data points is usually appropriate. For graphing two independent data arrays, certain parametric functions, or other points where there is not necessarily a logical connection between adjacent points, some kind of point or symbol graph is more appropriate. Several methods will be suggested in this section.

LINE GRAPH

The line graph as shown below is used to display data which one of the variables associated with each point is an orderly progression of values, such as would be generated by a function (Y = f(X)) evaluated at regular intervals throughout the X domain of interest. A line is also appropriate for graphing array data items versus their indices within an array.

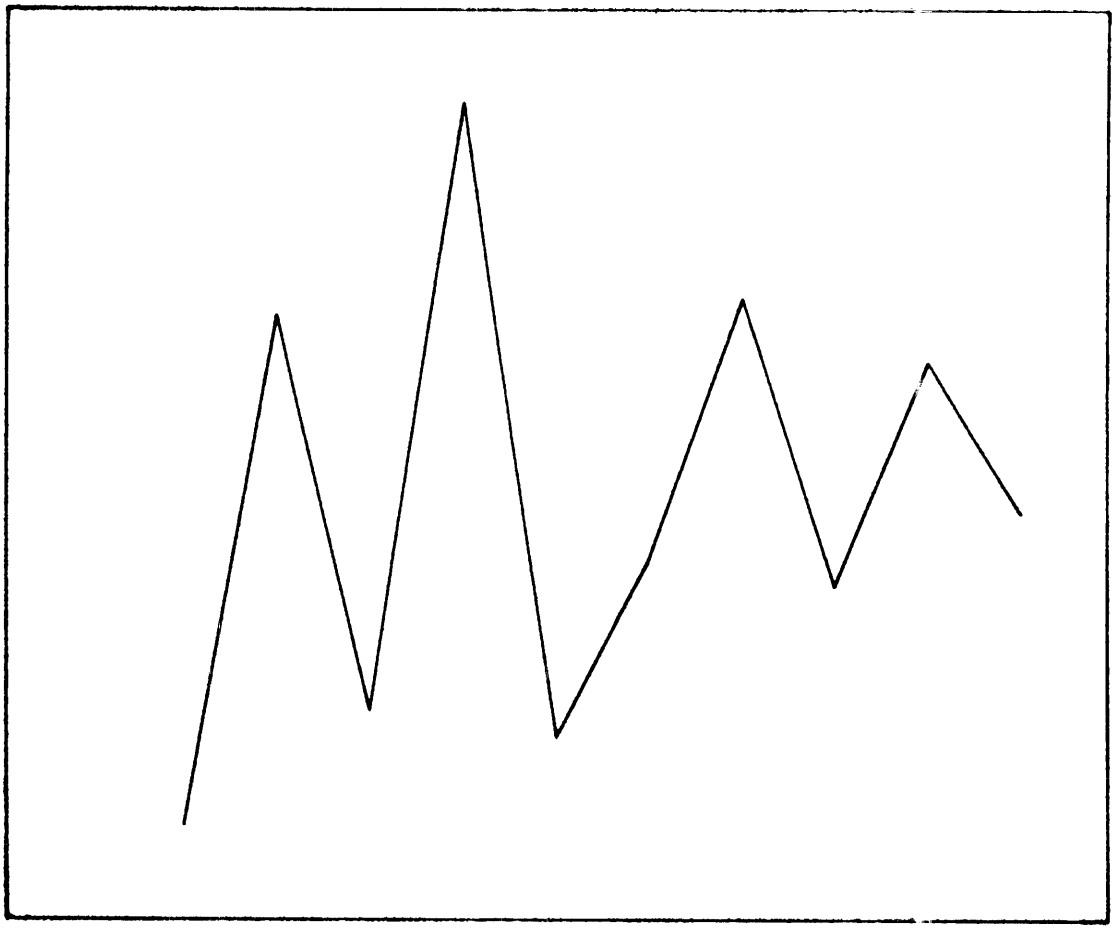
The line graph, or rather the actual data curve at the heart of a line graph, is generated by a series of DRAW commands. An important aspect of this process is that the DRAW command causes a line to be drawn from wherever the graphic point is currently located to the specified location within the window. When the first point on the data curve is drawn to, the graphic point is usually at some unknown or irrelevant location on the screen. So a MOVE to the first point to be graphed, rather than a DRAW, is appropriate. As a result, the FOR . . . NEXT loop which draws the remainder of the data starts with the second data point. A line graph of a one-dimensional array can be produced with the following statements (these must be run as part of a complete program):

```
      .
      .
      .
220 WINDOW 0,N,R1,R2
230 MOVE 1,Y(1)
240 FOR I=2 TO N
250 DRAW I,Y(I)
260 NEXT I
      .
      .
      .
```

N is the number of data values in the array Y.
R1 is the minimum data value in the array Y.
R2 is the maximum data value in the array Y.

A complete program to draw a line graph of ten random data values is shown below:

```
100 PAGE
110 INIT
120 N=10
130 DIM Y(N)
140 R1=1.0E+300
150 R2=-1.0E+300
160 FOR I=1 TO N
170 Y(I)=RND(-2)
180 R1=Y(I) MIN R1
190 R2=Y(I) MAX R2
200 NEXT I
210 VIEWPORT 10,120,10,90
220 WINDOW 0,N,R1,R2
230 MOVE 1,Y(1)
240 FOR I=2 TO N
250 DRAW I,Y(I)
260 NEXT I
270 END
```



A graph is sometimes easier to look at if it has some blank space around it. This is why the viewport is defined to be slightly smaller than full size in the above example (statement 210).

POINT AND SYMBOL GRAPH

Point Graph

When there is no direct relationship between adjacent data points, or no reason to connect them together with lines, a point graph or symbol graph is appropriate. The following example graphs two arrays. The first value in the X array determines the horizontal location of the first point, the first value in the Y array determines the vertical location of the first point, and so forth. (The statements below must be run as part of a larger program:)

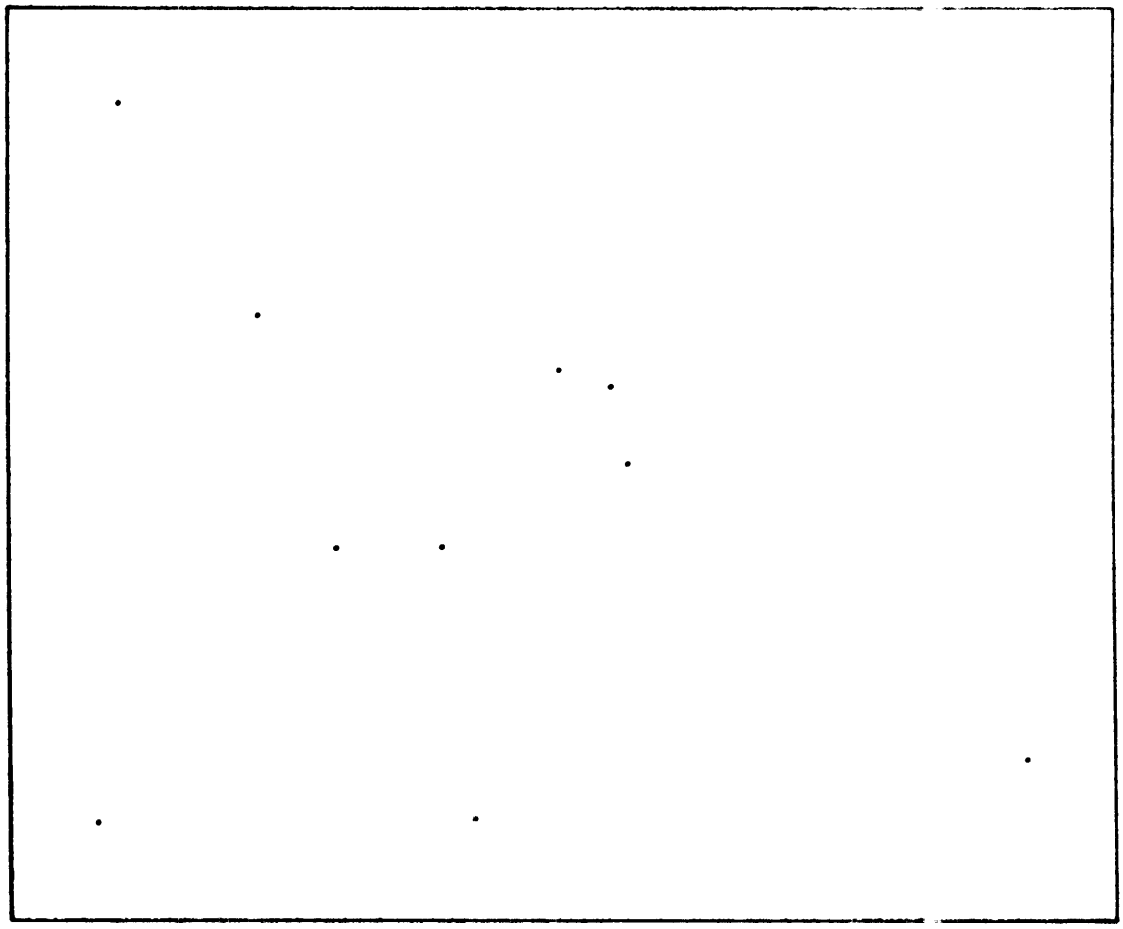
```
    .
    .
    .
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 DRAW X(I),Y(I)
290 NEXT I
    .
    .
    .
```

N is the number of data values in each array, R1 is the minimum value in the X array, R2 is the maximum value in the X array, S1 is the minimum value in the Y array and S2 is the maximum value in the Y array. In the FOR . . . NEXT loop, there is a MOVE to each point followed by a DRAW to each point. Since the graphic point is already located at the point drawn to, only a dot, not a line, is drawn.

A complete program to draw data points on the display is shown below:

```
100 PAGE
110 INIT
120 RESTORE
130 DATA 10,1.0E+300,-1.0E+300,1.0E+300,-1.0E+300
140 READ N,R1,R2,S1,S2
150 DIM X(N),Y(N)
160 FOR I=1 TO N
170 X(I)=RND(-2)
180 R1=X(I) MIN R1
190 R2=X(I) MAX R2
200 Y(I)=RND(-2)
210 S1=Y(I) MIN S1
220 S2=Y(I) MAX S2
230 NEXT I
240 VIEWPORT 10,120,10,90
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 DRAW X(I),Y(I)
290 NEXT I
300 END
```

This is the program's output:



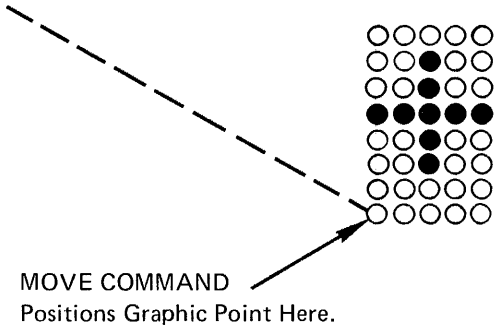
Printed Symbols

To make individual data points distinguishable, each point's location can be marked with a symbol instead of just a dot.

The simplest method is to use a standard GS character for the symbol. With this method, the graphic point is positioned on the display with a MOVE command (statement 270 in the examples above and below). Then the symbol is written onto the display with a PRINT command, as shown in the program fragment below (these statements must be run as part of a larger program):

```
    .
    .
    .
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 PRINT "+";
290 NEXT I
    .
    .
    .
```

Although the character "+" is shown in the example, any character can be used. If these statements (above) are used, the center of the printed character will be above and to the right of the desired location on the display (see diagram below).



The above diagram shows that the MOVE command positions the lower left corner of the character. If the center of the printed character is to coincide with the point specified by the MOVE command, the graphic point must be re-positioned slightly after the MOVE command (statement 270) and before the PRINT command (statement 280). The statements added in the following example (273, 278, and 285) center the "+" over the graphic point.

```
    .
    .
    .
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
273 SCALE 1,1
278 RMOVE -0.5*1.55,-0.5*1.88
280 PRINT "+";
282 RMOVE 0.5*1.55,0.5*1.88
285 WINDOW R1,R2,S1,S2
290 NEXT I
    .
    .
    .
```

Since the characters written by the PRINT command are always the same size, the graphic point must always be re-positioned the same distance, regardless of how the window has been defined. The SCALE 1,1 command at statement 273 defines a new data space to fill the viewport. Within this new data space, 1 data unit is the same length as 1 GDU. The RMOVE command is then used to re-position the graphic point relative to its current position. In order to center the character, the graphic point should be moved half a character width to the left and half a character height down. The RMOVE command at statement 278 performs this function. After the character is printed, the window must be restored to its previously defined limits. If it is not restored, the remainder of the graphed points will be placed inaccurately on the display.

Care must be exercised if the above technique (PRINTed characters as symbols) is used when a default (full size) viewport has been defined. If the graphic point is at the very top, bottom, left or right edge of the physical display when the character is printed, the character will not be printed in the correct location. This inaccuracy is prevented whenever the defined viewport is less than full size. The largest viewport which still prevents this inaccuracy from occurring is defined with the following command:

```
VIEWPORT 2,128,2,98.
```

Drawn Symbols

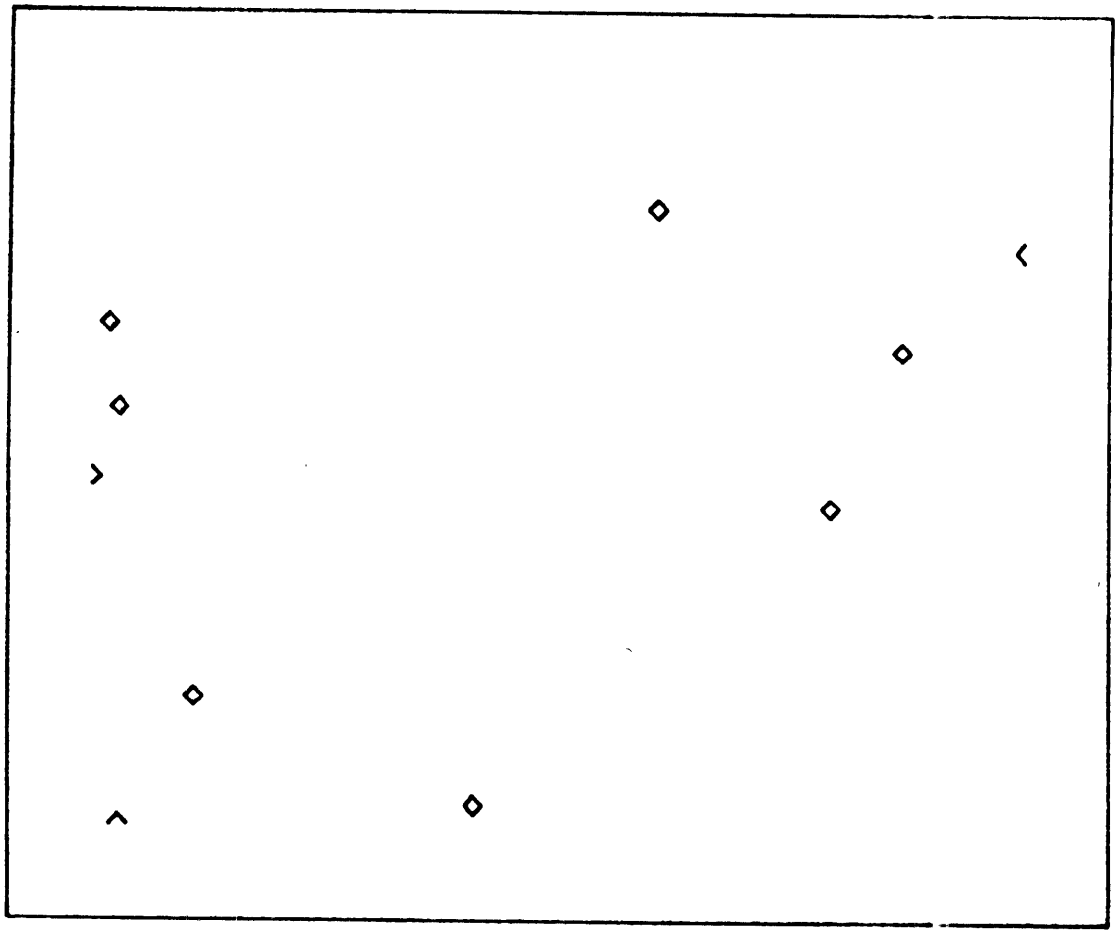
Another way to place a symbol at the location desired is to actually draw it. This is done with a series of relative moves and draws from the location the symbol indicates. The following example draws a diamond to indicate a location:

```
      .
      .
      .
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 SCALE 1,1
290 RMOVE 1,0
300 RDRAW -1,-1
310 RDRAW -1,1
320 RDRAW 1,1
330 RDRAW 1,-1
340 RMOVE -1,0
350 WINDOW R1,R2,S1,S2
360 NEXT I
      .
      .
      .
```

The SCALE 1,1 at statement 280 ensures that the symbols will always be the same size regardless of how the window and viewport have been defined. Line 290 ensures that the symbol is centered over the location to be indicated. Lines 300 through 330 draw the symbol. Line 340 restores the graphic point to its location before the symbol was drawn. Line 350 restores the window so that subsequent data points will be graphed correctly.

Below is the complete program and the output it produces.

```
100 PAGE
110 INIT
120 RESTORE
130 DATA 10,1.0E+300,-1.0E+300,1.0E+300,-1.0E+300
140 READ N,R1,R2,S1,S2
150 DIM X(N),Y(N)
160 FOR I=1 TO N
170 X(I)=RND(-2)
180 R1=X(I) MIN R1
190 R2=X(I) MAX R2
200 Y(I)=RND(-2)
210 S1=Y(I) MIN S1
220 S2=Y(I) MAX S2
230 NEXT I
240 VIEWPORT 10,120,10,90
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 SCALE 1,1
290 RMOVE 1,0
300 RDRAW -1,-1
310 RDRAW -1,1
320 RDRAW 1,1
330 RDRAW 1,-1
340 RMOVE -1,0
350 WINDOW R1,R2,S1,S2
360 NEXT I
370 END
```



Drawn symbols are subject to clipping. Printed symbols are not.

The symbol size can be made proportional to a data value or function result as shown in the following program fragment:

```
    .
    .
    .
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 SCALE 1,1
290 RMOVE Z(I),0
300 RDRAW -Z(I),-Z(I)
310 RDRAW -Z(I),Z(I)
320 RDRAW Z(I),Z(I)
330 RDRAW Z(I),-Z(I)
340 RMOVE -Z(I),0
350 WINDOW R1,R2,S1,S2
360 NEXT I
    .
    .
    .
```

More than one kind of symbol can be drawn by a program. In addition, the actual symbol can be selected by a data value. Using the GOSUB . . . OF . . . command, as shown below, makes this very easy.

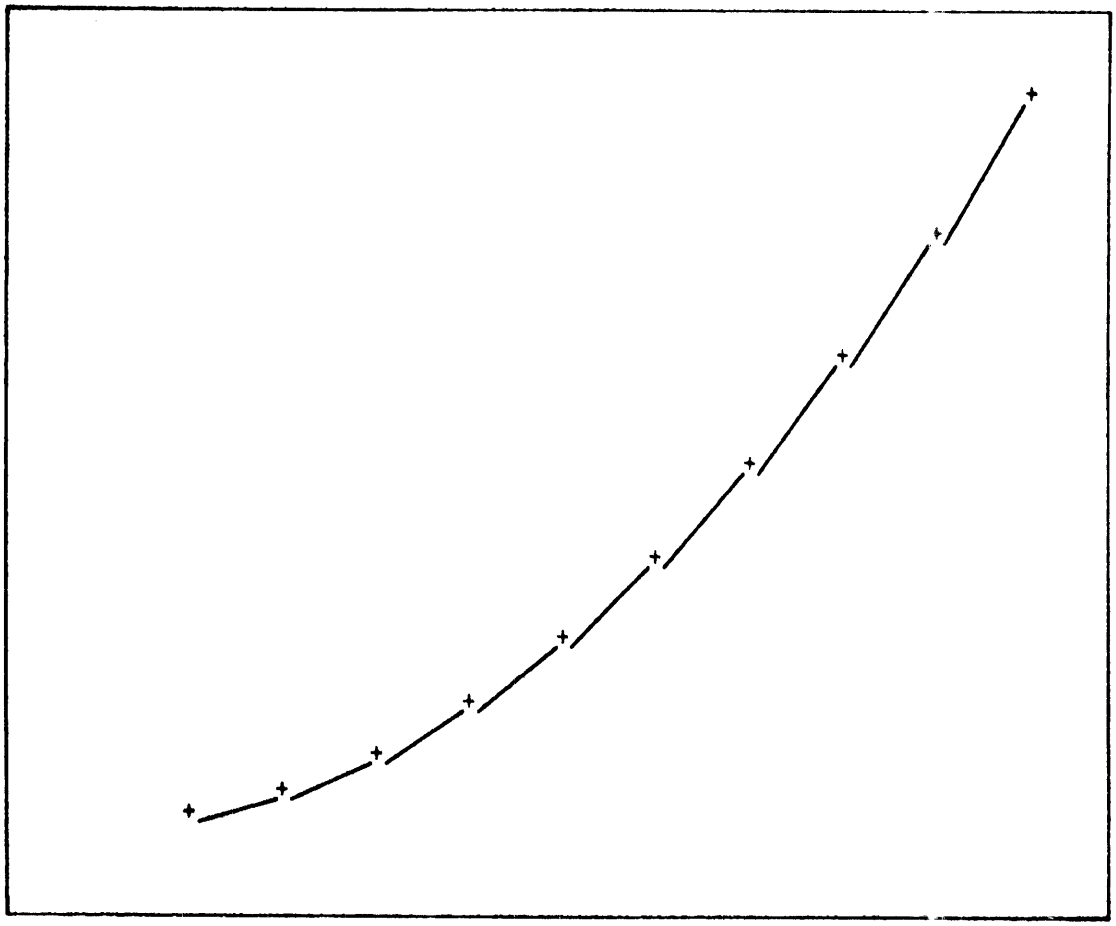
```
    .
    .
    .
250 WINDOW R1,R2,S1,S2
260 FOR I=1 TO N
270 MOVE X(I),Y(I)
280 SCALE 1,1
290 GOSUB Z(I) OF 500,600,700
300 WINDOW R1,R2,S1,S2
310 NEXT I
320 END
    .
    .
    .
500 REM  DRAW A DIAMOND SYMBOL
    .
    .
    .
550 RETURN
600 REM  DRAW A SQUARE SYMBOL
    .
    .
    .
650 RETURN
700 REM  DRAW A TRIANGLE SYMBOL
    .
    .
    .
750 RETURN
    .
    .
    .
```

MULTI-LINE GRAPHS

If there are several lines on one graph, a very good technique to identify each line is to draw a symbol at each graphed point. A program like this:

```
    .
    .
    .
220 WINDOW 0,N,R1,R2
230 MOVE 1,A(1)
240 GOSUB 400
250 FOR I=2 TO N
260 DRAW I,A(I)
270 GOSUB 400
280 NEXT I
    .
    .
    .
400 REM  PRINT SYMBOL
410 PRINT "+";
420 RETURN
    .
    .
    .
```

will produce a line like this:

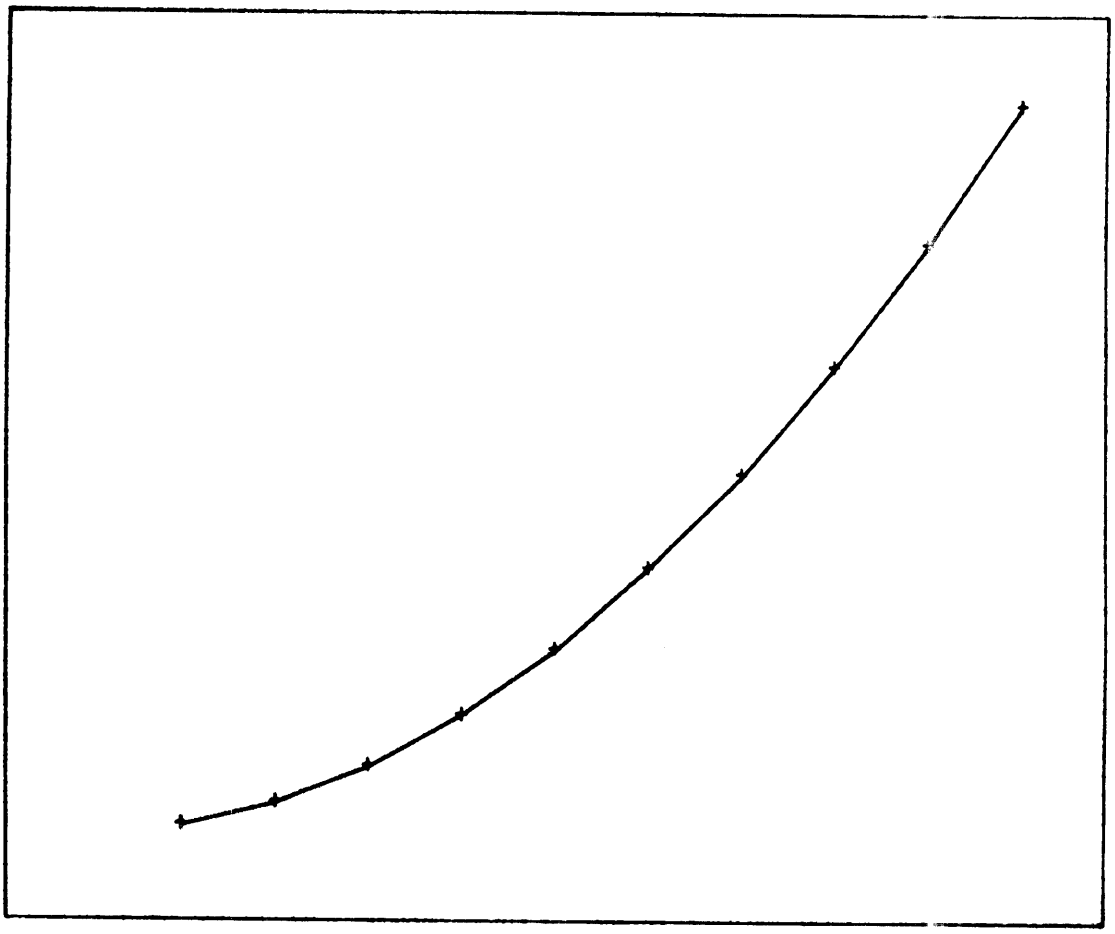


To properly position the printed character, the correction described on page 3-6 must be included. The corrected program segment is shown below:

```
      .
      .
      .
220 WINDOW 0,N,R1,R2
230 MOVE 1,A(1)
240 GOSUB 400
250 FOR I=2 TO N
260 DRAW I,A(I)
270 GOSUB 400
280 NEXT I
      .
      .
      .
400 REM  PRINT SYMBOL WITH CORRECTED POSITION
405 SCALE 1,1
410 RMOVE -0.5*1.55,-0.5*1.88
420 PRINT "+";
430 RMOVE 0.5*1.55,0.5*1.88
440 WINDOW 0,N,R1,R2
450 RETURN
      .
      .
      .
```

Below is a complete program which uses the above technique. The program output is shown also.

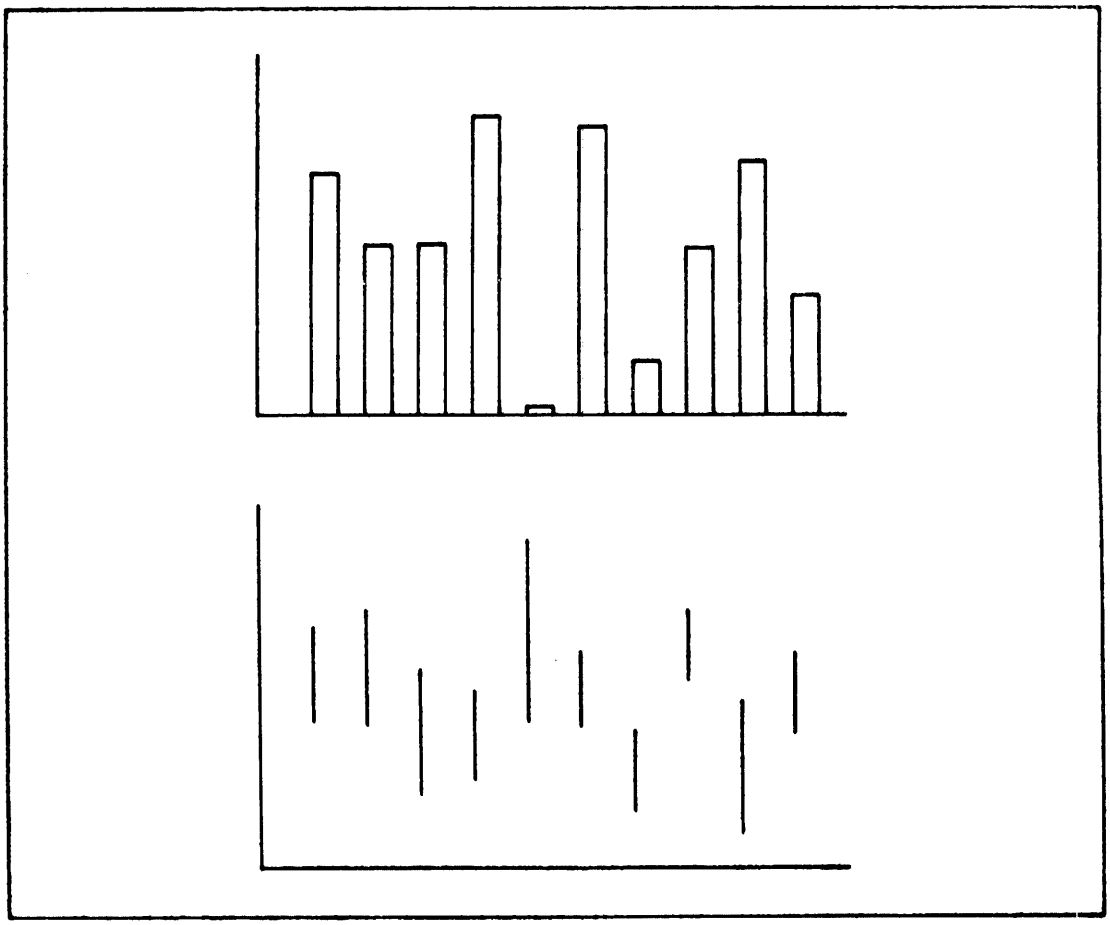
```
100 PAGE
110 INIT
120 RESTORE
130 DATA 10,1.0E+300,-1.0E+300
140 READ N,R1,R2
150 DIM A(N)
160 FOR I=1 TO N
170 A(I)=I↑2
180 R1=A(I) MIN R1
190 R2=A(I) MAX R2
200 NEXT I
210 VIEWPORT 10,120,10,90
220 WINDOW 0,N,R1,R2
230 MOVE 1,A(1)
240 GOSUB 400
250 FOR I=2 TO N
260 DRAW I,A(I)
270 GOSUB 400
280 NEXT I
290 END
400 REM  PRINT SYMBOL WITH CORRECTED POSITION
405 SCALE 1,1
410 RMOVE -0.5*1.55,-0.5*1.88
420 PRINT "+";
430 RMOVE 0.5*1.55,0.5*1.88
440 WINDOW 0,N,R1,R2
450 RETURN
```



Even with this correction included, the characters are not positioned with their centers exactly over the desired point. If the position of the symbol must precisely coincide with the indicated location, drawn symbols are more appropriate than printed symbols.

OTHER TYPES OF GRAPHS

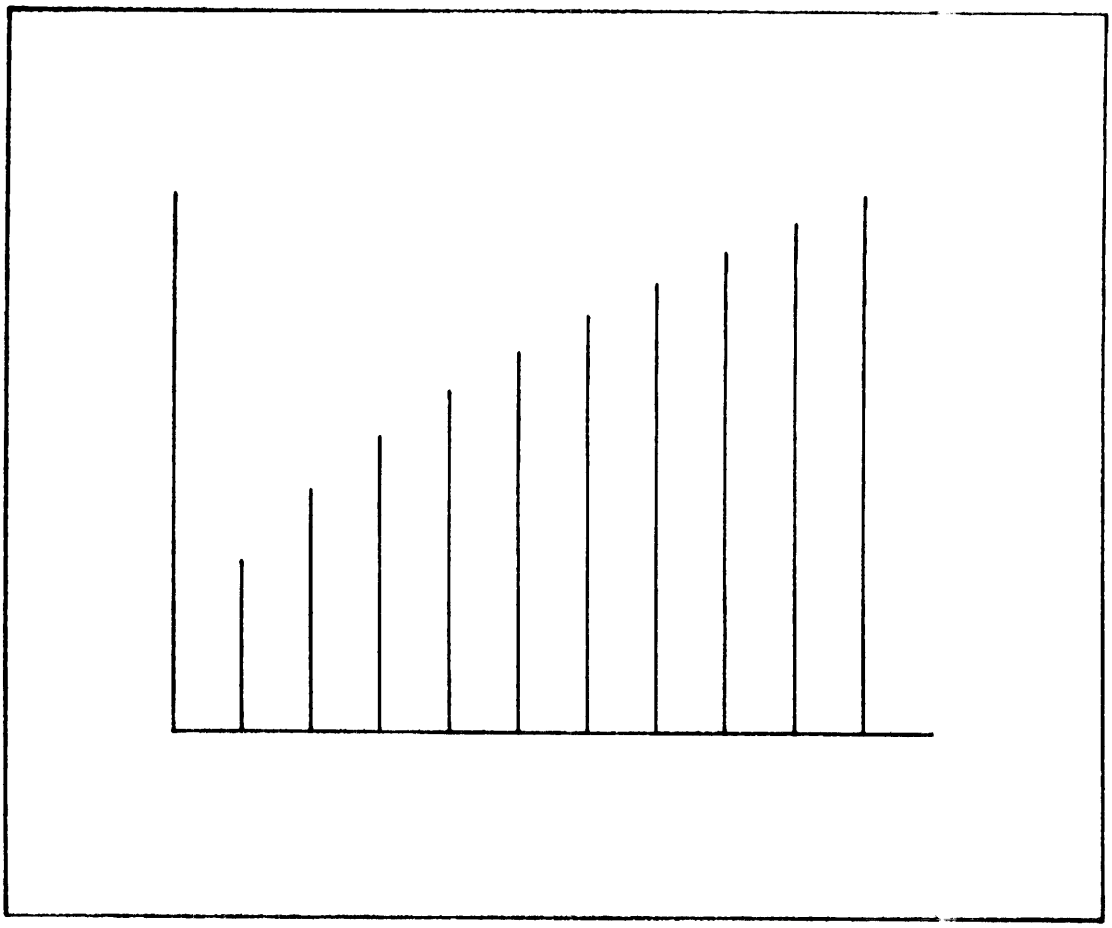
Another way to represent certain kinds of data is with vertical or horizontal rectangles instead of lines and points.



These are called bar charts or histograms. They are really a special type of symbol graph. The symbol in this case is a rectangle or line whose height is proportional to the data value being represented. One end of the rectangle or line is usually on a base line (as in the first example graph, above). However, depending upon the data being presented, this will not always be the case (as in the second example graph). This second technique is sometimes used to present the performance range of a variable at a given time, such as daily stock market averages.

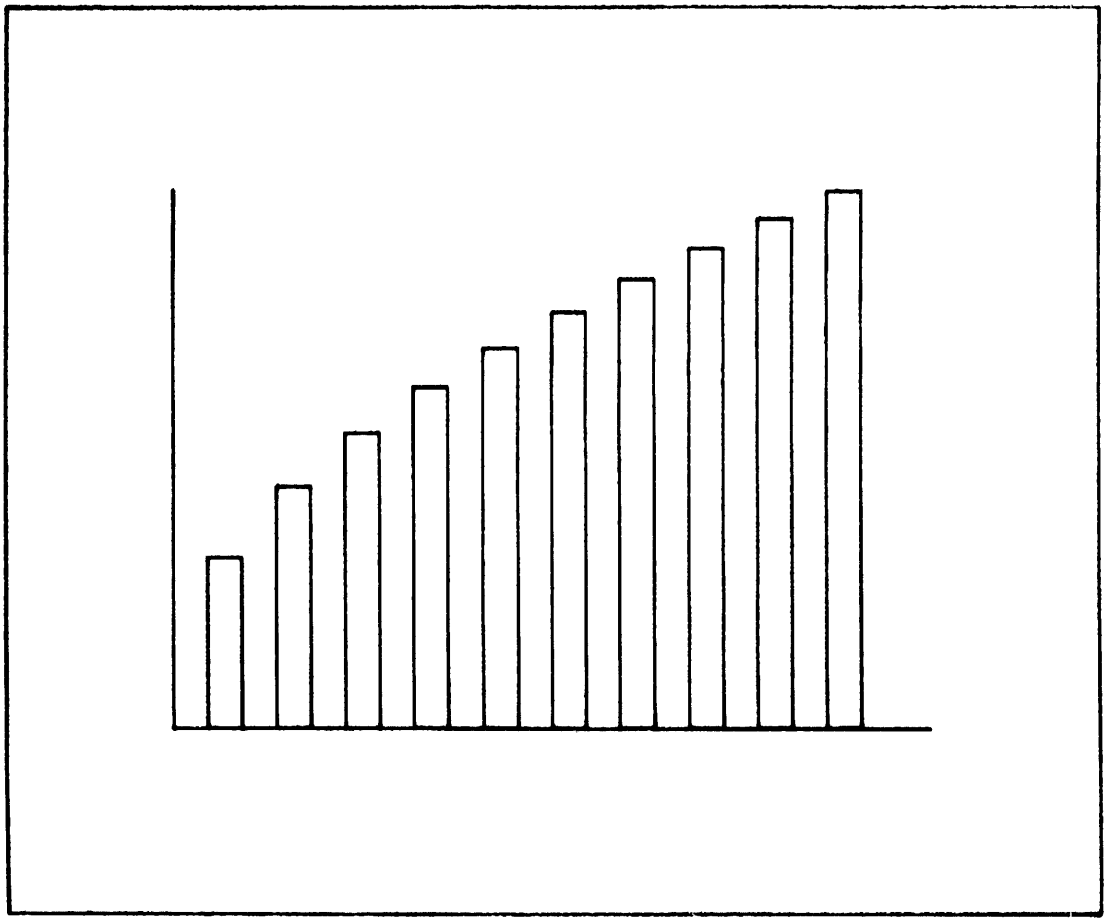
Here is a way to draw vertical lines whose lengths represent data values:

```
      .
      .
      .
200 WINDOW 0,N*1.1,0,R
210 AXIS
220 FOR I=1 TO N
230 MOVE I,0
240 DRAW I,A(I)
250 NEXT I
      .
      .
      .
      .
      .
      .
```



The vertical lines can be changed to vertical bars by replacing the DRAW statement at statement 240 with a subroutine branch.

```
    .
    .
    .
200 WINDOW 0,N*1.1,0,R
210 AXIS
220 FOR I=1 TO N
230 MOVE I,0
240 GOSUB 400
250 NEXT I
    .
    .
    .
400 RDRAW -0.5,0
410 RDRAW 0,A(I)
420 RDRAW 0.5,0
430 RDRAW 0,-A(I)
440 RETURN
    .
    .
    .
```



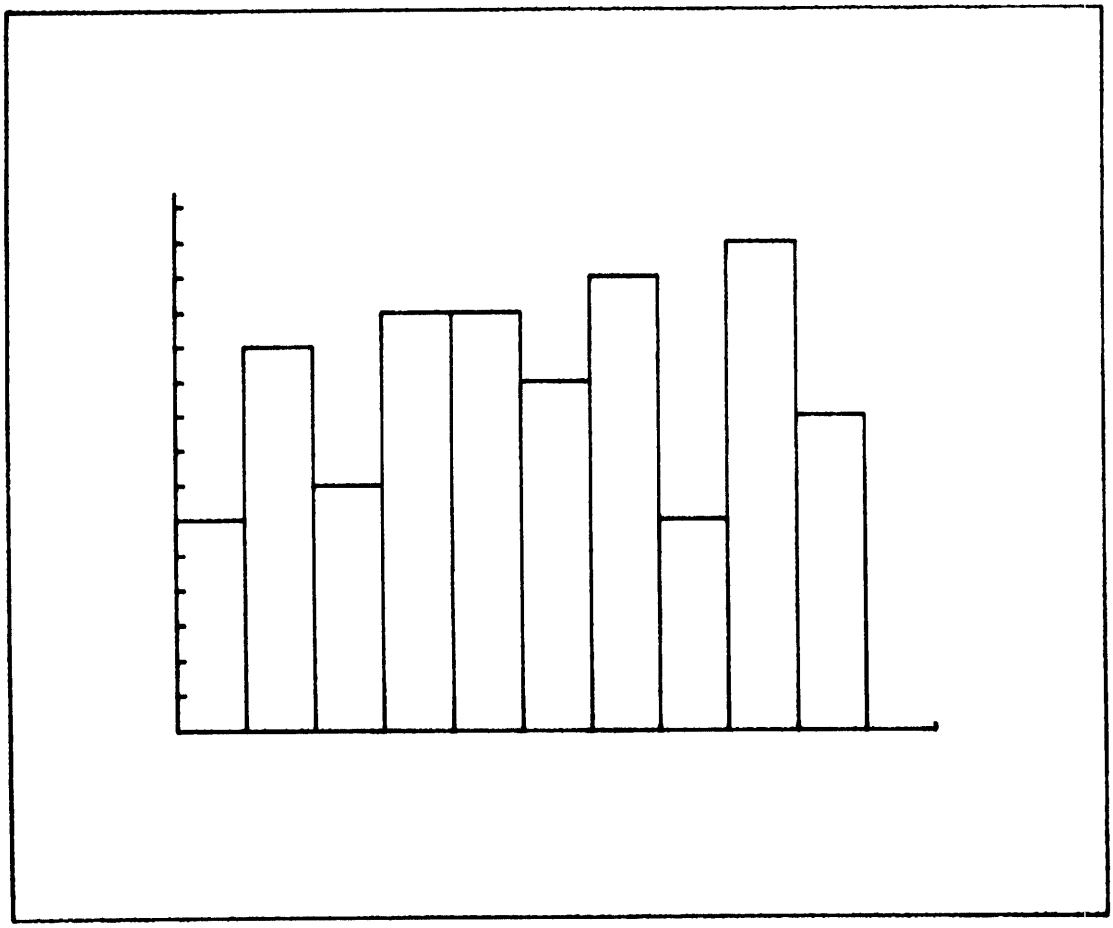
The heights of the bars are the same as the heights of the lines in the previous example.

Here is the complete program which generated the output shown above:

```
100 PAGE
110 INIT
120 N=10
130 R=-1.0E+300
140 DIM A(N)
150 FOR I=1 TO N
160 A(I)=SQR(I)
170 R=A(I) MAX R
180 NEXT I
190 VIEWPORT 20,110,20,80
200 WINDOW 0,N*1.1,0,R
210 AXIS
220 FOR I=1 TO N
230 MOVE I,0
240 GOSUB 400
250 NEXT I
260 END
400 RDRAW -0.5,0
410 RDRAW 0,A(I)
420 RDRAW 0.5,0
430 RDRAW 0,-A(I)
440 RETURN
```

In the previous example, the height of each bar represented a value contained in a specific array item, i.e., the value of one variable. Often this bar graph technique will be used to graph a continuous variable, such as the distribution of 50 occurrences of a random number which can have any value between 0 and 1. The next example shows how th s can be done.

```
100 PAGE
110 INIT
120 S=100
130 R=-1.0E+300
140 DIM A(10)
150 A=0
160 FOR I=1 TO S
170 J=INT(RND(-2)*10)+1
180 A(J)=A(J)+1
190 R=A(J) MAX R
200 NEXT I
210 VIEWPORT 20,110,20,80
220 WINDOW 0,11,0,R*1.1
230 AXIS 1,1
240 FOR I=1 TO 10
250 MOVE I,0
260 GOSUB 400
270 NEXT I
280 END
400 RDRAW -1,0
410 RDRAW 0,A(I)
420 RDRAW 1,0
430 RDRAW 0,-A(I)
440 RETURN
```



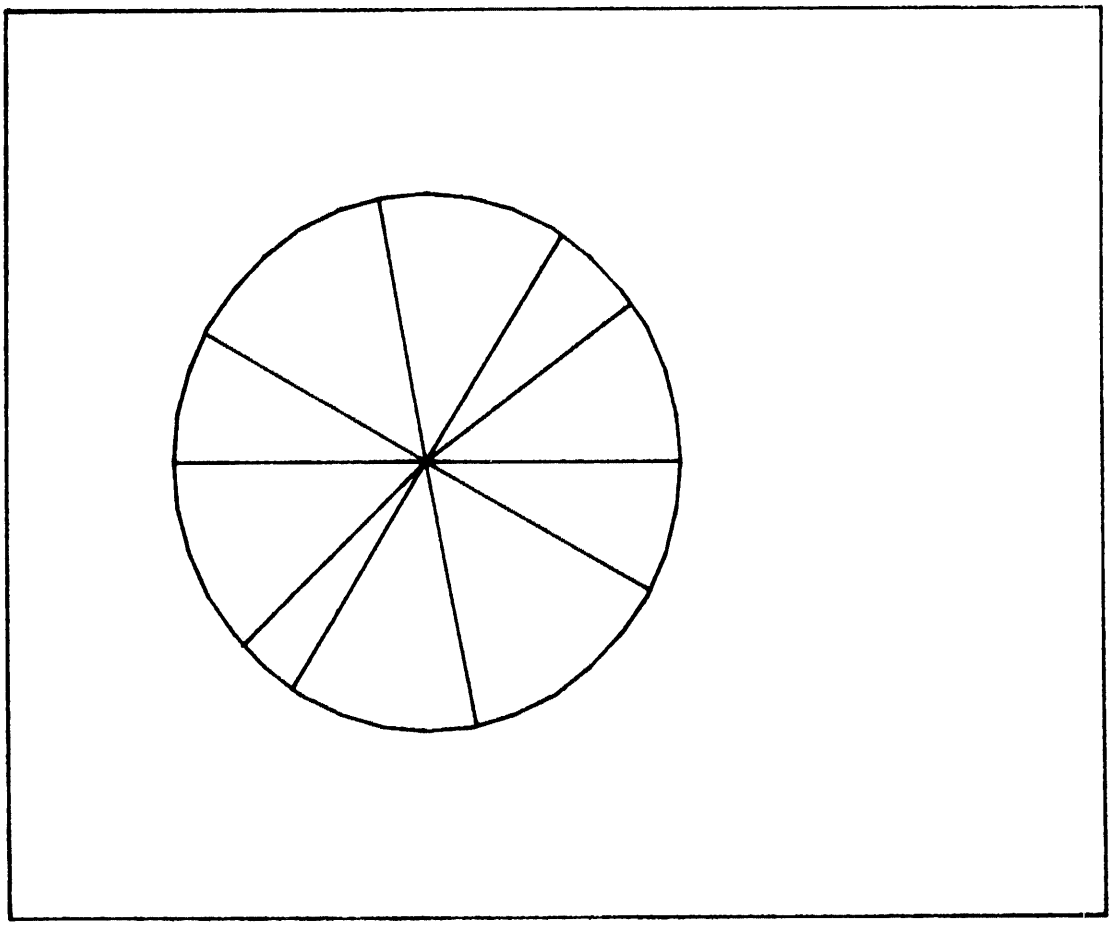
The loop which actually does the graphing (statements 240 through 270) is similar to previous examples. The width of the bars has been increased until the bars are now adjacent to each other. The array A is filled by statements 160 through 200. A(1) contains the quantity of random numbers whose values fell between 0 and .1, A(2) contains the quantity of random numbers whose values fell between .1 and .2, and so forth. The MAX function is useful to find the maximum number of incidences in any category (statement 190). The resulting maximum value (R) is used in the WINDOW command in statement 220.

Graphing the percentage each category is of the total number of incidences (50 in this example) requires only a vertical rescaling determined by the data. The quantity of incidences in each category is divided by the total number of incidences to yield a decimal fraction. This is multiplied by 100 to yield a percentage. The graph looks the same, so only the listing is shown as the example.

```
100 PAGE
110 INIT
120 S=50
130 R=-1.0E+300
140 DIM A(10)
150 A=0
160 FOR I=1 TO S
170 J=INT(RND(-2)*10)+1
180 A(J)=A(J)+1
190 R=A(J) MAX R
200 NEXT I
210 VIEWPORT 20,110,20,80
220 WINDOW 0,11,0,R/50*100*1.1
230 AXIS 1,5
240 FOR I=1 TO 10
250 MOVE I-1,0
260 GOSUB 400
270 NEXT I
280 END
400 RDRAW 0,A(I)/50*100
410 RDRAW 1,0
420 RDRAW 0,-A(I)/50*100
430 RETURN
```

Data expressed in this percentage form can be graphed in another way. The height of each vertical bar, now a percentage, can be interpreted to be the width of a slice of a circle. This produces a "pie chart", shown in the following example. The same data is being used.

```
100 PAGE
110 INIT
120 S=50
130 R=-1.0E+300
140 DIM A(10)
150 A=0
160 FOR I=1 TO S
170 J=INT(RND(-2)*10)+1
180 A(J)=A(J)+1
190 R=A(J) MAX R
200 NEXT I
210 VIEWPORT 20,80,20,80
220 WINDOW -1,1,-1,1
225 SET DEGREES
230 MOVE 1,0
240 FOR I=1 TO 10
242 FOR T=10 TO 360 STEP 10
244 DRAW COS(T),SIN(T)
246 NEXT T
250 B=0
260 FOR I=1 TO 10
270 B=B+360*(A(I)/50)
280 MOVE 0,0
290 DRAW COS(B),SIN(B)
300 NEXT I
310 END
```



Section 4

TRANSFORMATIONS

INTRODUCTION

Up to this point, the discussion has covered graphs where the actual function or data is presented. None of the data has been manipulated between entry and graphing. However, there will be many instances where the actual raw data or function will not be of interest unless it is suitably transformed. Three examples:

temperature data taken in Fahrenheit degrees but used in a formula which requires Celsius degrees;

data taken in miles per hour but used in the form of meters per second;

data taken in the form of dollars per hundred weight but used in the form of francs per kilogram.

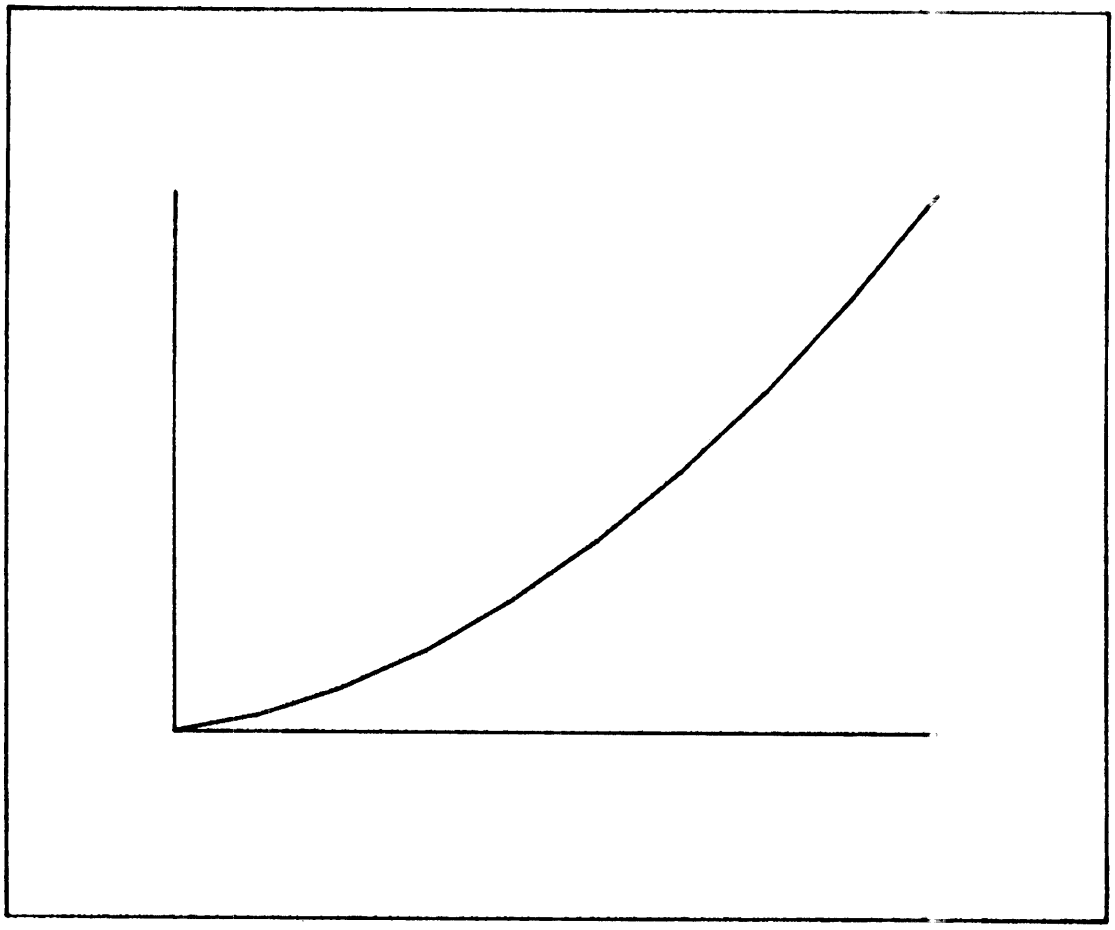
There will be many situations where the transformation is neither linear nor simple. For example, in a laboratory situation, temperature might be measured with a resistor whose resistance varies as a function of its surrounding temperature. Thus the instrumentation which supplies data to the GS will be sending data values in the form of ohms. The GS program handles data in the form of degrees Celsius. The formula for the relationship between ohms and degrees Celsius over the range of interest is used as a transformation. During data editing and graphing, the data being dealt with will then be degrees, not ohms.

EXAMPLES

In the three examples which follow, lines 100 through 210 are identical. They fill array A with data values and place the smallest and largest of these values into M1 and M2 respectively. And, in all three examples, lines 250 through 280 perform identical functions: graphing the data in array A.

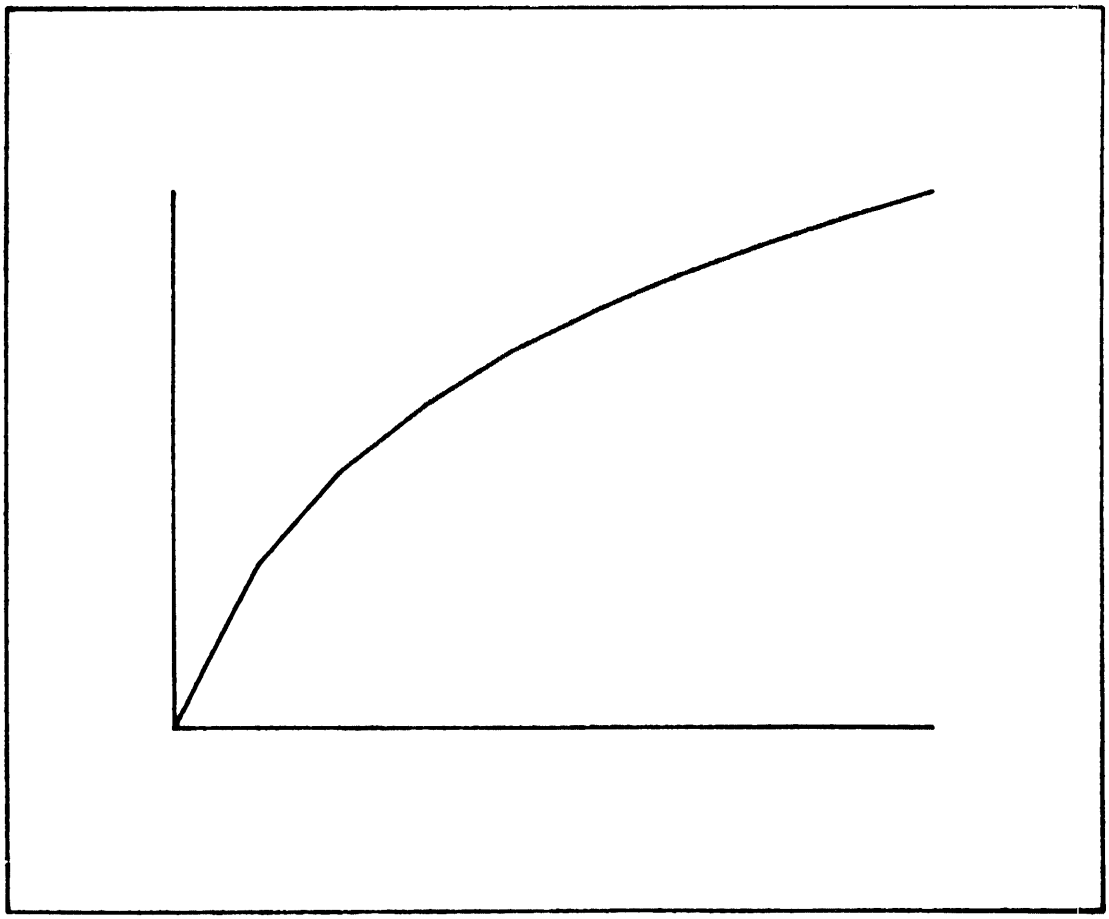
In the first example, the data is graphed with the index of the array item on the horizontal axis and the actual array value on the vertical. Both scales are linear.

```
100 PAGE
110 INIT
120 VIEWPORT 20,110,20,80
130 N=10
140 DIM A(N)
150 M1=1.0E+300
160 M2=-1.0E+300
170 FOR I=1 TO N
180 A(I)=I↑2
190 M1=M1 MIN A(I)
200 M2=M2 MAX A(I)
210 NEXT I
220 REM  GRAPH ACTUAL DATA (HORIZ & VERT.)
230 WINDOW 1,N,M1,M2
240 AXIS
250 MOVE 1,A(1)
260 FOR I=2 TO N
270 DRAW I,A(I)
280 NEXT I
290 END
```



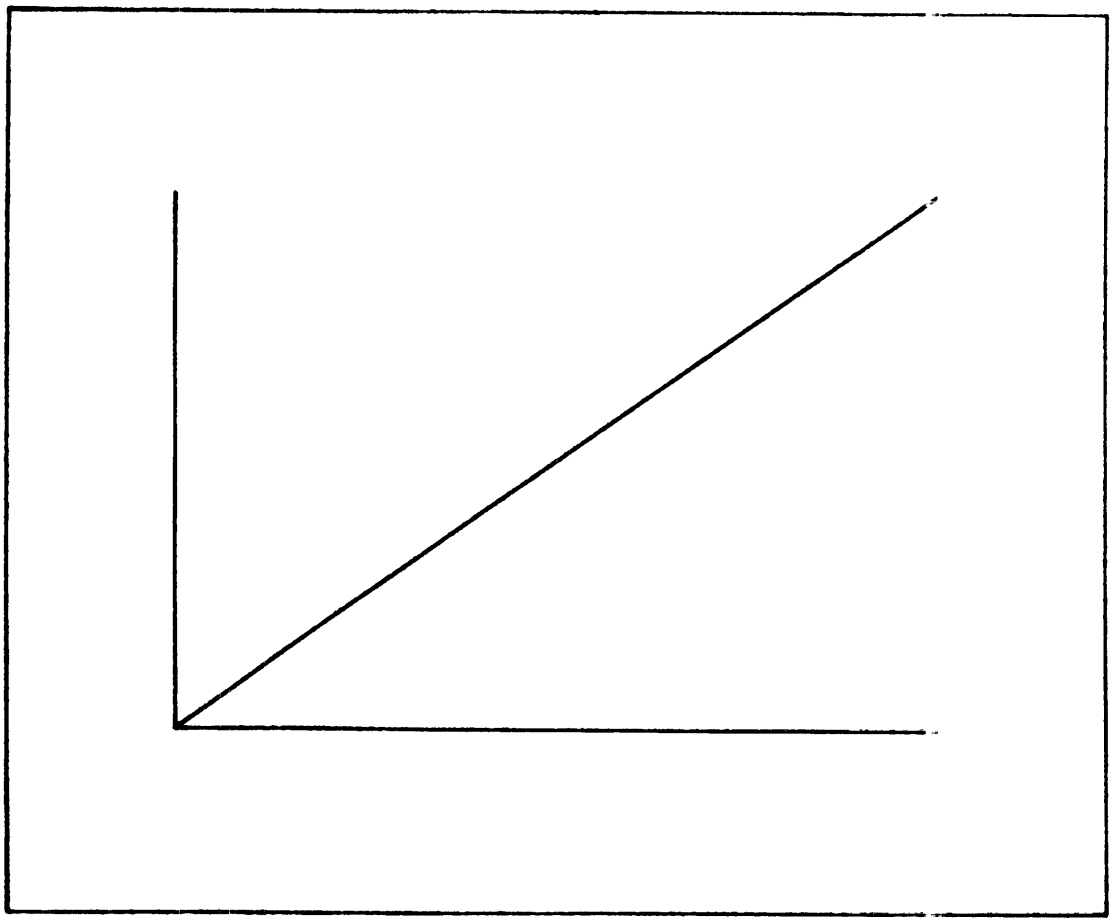
In the next example, the horizontal information is as before, the index of the array item. The vertical information is not the actual value contained in the array item but the log to the base 10 of the value in each array item.

```
100 PAGE
110 INIT
120 VIEWPORT 20,110,20,80
130 N=10
140 DIM A(N)
150 M1=1.0E+300
160 M2=-1.0E+300
170 FOR I=1 TO N
180 A(I)=I↑2
190 M1=M1 MIN A(I)
200 M2=M2 MAX A(I)
210 NEXT I
220 REM  VERTICAL DATA TRANSFORMED BY LOG BASE 10
230 WINDOW 1,N,LGT(M1),LGT(M2)
240 AXIS
250 MOVE 1,LGT(A(1))
260 FOR I=2 TO N
270 DRAW I,LGT(A(I))
280 NEXT I
290 END
```



In the third example, the logs of both the array item index and the value contained in the array item are graphed.

```
100 PAGE
110 INIT
120 VIEWPORT 20,110,20,80
130 N=10
140 DIM A(N)
150 M1=1.0E+300
160 M2=-1.0E+300
170 FOR I=1 TO N
180 A(I)=I↑2
190 M1=M1 MIN A(I)
200 M2=M2 MAX A(I)
210 NEXT I
220 REM  BOTH HORIZ. & VERT. DATA TRANSFORMED BY LOG BASE 10
230 WINDOW LGT(1),LGT(N),LGT(M1),LGT(M2)
240 AXIS
250 MOVE LGT(1),LGT(A(1))
260 FOR I=2 TO N
270 DRAW LGT(I),LGT(A(I))
280 NEXT I
290 END
```



In all of the previous examples, the type of transformation used is reflected in three areas:

* In the WINDOW command (line 230 in each example). The transformation is used to make sure that the data completely fills the specified viewport. Because each example uses a different transformation, the viewport in each case is "looking at" slightly different areas of the user's data space. As a result of the transformation being reflected in the WINDOW statement, the starting and ending points of all three data curves are in the same place on each graph.

* In the MOVE command (line 250 in each example). As described previously, this statement is necessary so that the data curve will start at the first data point, not at wherever the graphic point is when the program is run. The transformation must appear here for the data curve to start in the appropriate location.

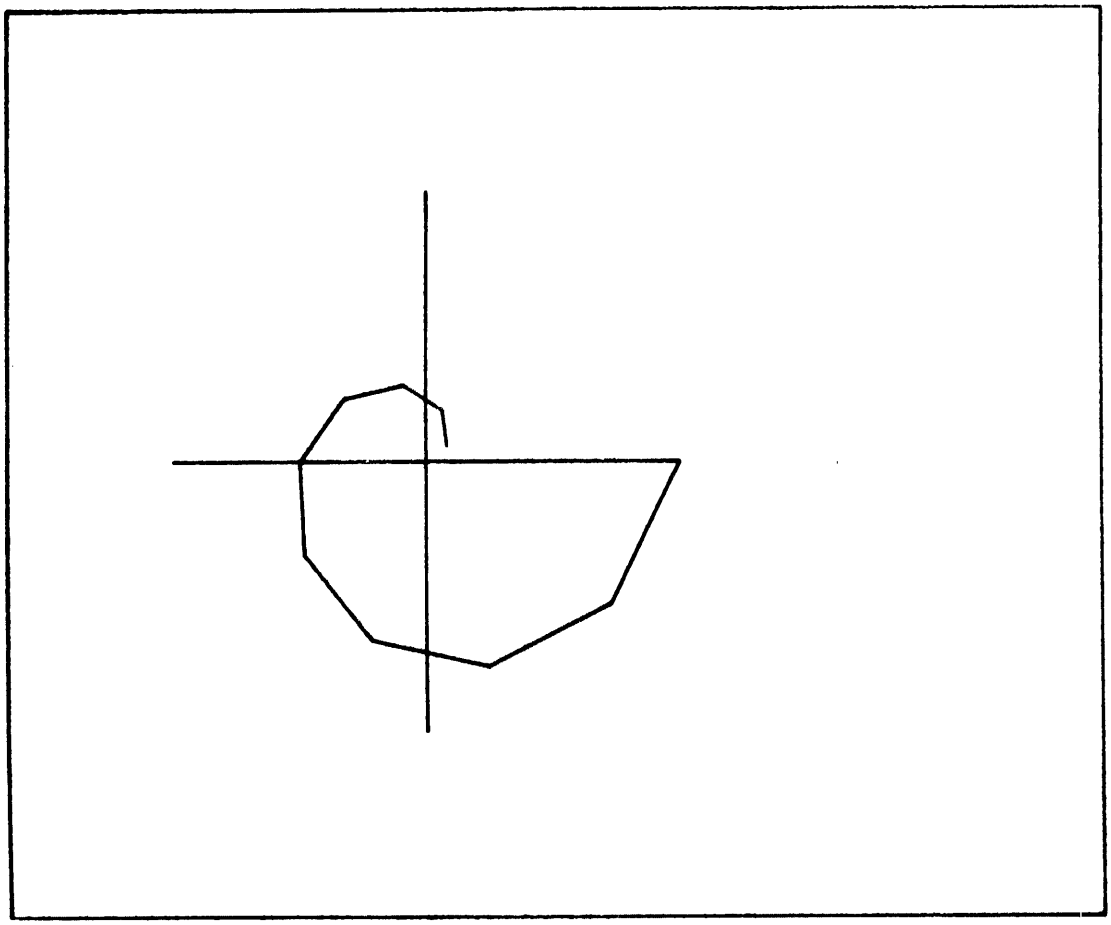
* In the DRAW command (line 270 in each example). Since this is the statement which actually causes the data curve to be drawn, the transformation must appear here so that the curve is drawn as desired.

TWO APPROACHES

There are two significantly different approaches to this transformation process. In one approach, the data is stored and edited in the GS in its original incoming form. The data is transformed to a new form only when it is graphed. This approach is appropriate if the data in its incoming form is the primary concern. The way in which it is graphed is of secondary interest only. This approach was used in the group of three examples. A polar graph is another example. It is always appropriate to deal with polar data in its original form of angle and radius. But to graph it on the GS, it must be transformed into cartesian coordinates. As in the three examples, this transformation must be reflected in the WINDOW command (statement 240, below) in the initial move to the starting point of the data curve (statement 270, below), and in the DRAW command which actually draws the curve (statement 290, below).

In a polar graph, as in some other types, the transformation used must also be reflected in the minimum-maximum determination (statements 190, 200, and 230, below). This is because the data minimum and maximum do not coincide with minimum and maximum horizontal and vertical locations on the screen. (This is the case in the three examples.)

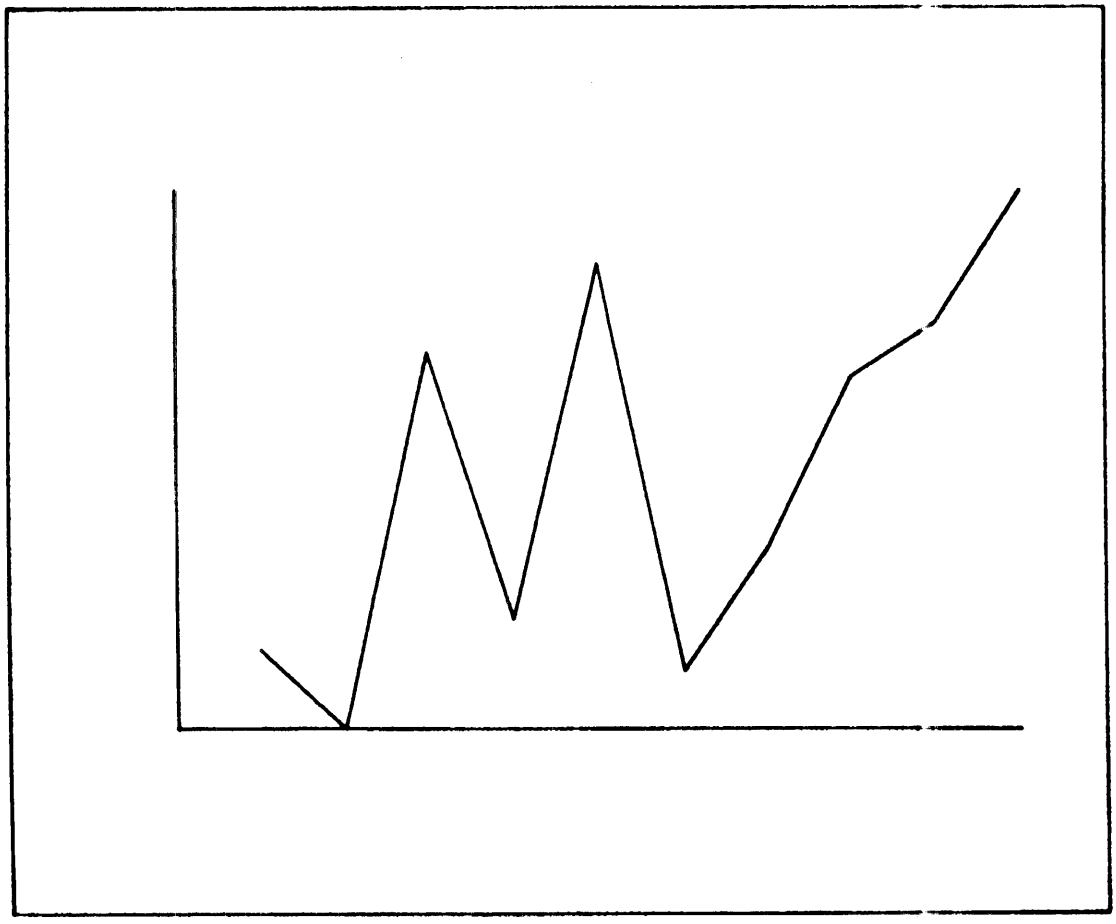
```
100 PAGE
110 INIT
120 SET DEGREES
130 DIM R(10),T(10)
140 M1=1.0E+300
150 M2=-1.0E+300
160 FOR I=1 TO 10
170 T(I)=36*I
180 R(I)=I
190 M1=M1 MIN R(I)
200 M2=M2 MAX R(I)
210 NEXT I
220 REM  NOW FIND MAXIMUM RADIUS
230 M3=ABS(M1) MAX ABS(M2)
240 WINDOW -M3,M3,-M3,M3
250 VIEWPORT 20,80,20,80
260 AXIS
270 MOVE R(1)*COS(T(1)),R(1)*SIN(T(1))
280 FOR I=2 TO 10
290 DRAW R(I)*COS(T(I)),R(I)*SIN(T(I))
300 NEXT I
310 END
```



Polar graphs have another special characteristic. They require equal horizontal and vertical scaling. This means that a vertical user data unit must have the same displayed length as a horizontal user data unit. If this is not the case, objectionable distortion will result. For example, a circle might be drawn as an ellipse. This requirement can be satisfied by using the SCALE command with equal arguments. Another solution, shown in the polar graph example, is to make sure that the window and viewport have the same aspect ratio. This is another way of saying that their shapes are proportional. In the above example, both the window and viewport are square. As a result, the spiral data curve is undistorted.

In the second approach to the transformation process, the data is transformed prior to storage. This approach is desirable when the form in which the original data is received is never appropriate or convenient to use (such as when the incoming data is in the form of degrees Fahrenheit and is to be used in the form of degrees Celsius). In this approach, the actual transformation is done when the data is input. This means that the transformation needs to be used only when the data is transformed for storage (statement 210 in the example below). The min-max determination, the WINDOW and the MOVE to the first point all are performed on the transformed data. This approach is less cumbersome than transforming the data just prior to graphing and is more convenient to use when the situation allows.

```
100 PAGE
110 INIT
120 VIEWPORT 20,120,20,80
130 N=10
140 M1=1.0E+300
150 M2=-1.0E+300
160 DIM C(N)
170 FOR I=1 TO N
180 REM  STATEMENT 190 SIMULATES INPUT @D:
190 F=RND(-2)*180+32
200 REM  CONVERT INPUT DATA FROM FAHRENHEIT TO CELSIUS
210 C(I)=5*(F-32)/9
220 M1=M1 MIN C(I)
230 M2=M2 MAX C(I)
240 NEXT I
250 WINDOW 0,N,M1,M2
260 AXIS
270 MOVE 1,C(1)
280 FOR I=2 TO N
290 DRAW I,C(I)
300 NEXT I
310 END
```



USEFUL TYPES

Occasionally, the transform which operates on the incoming data must be variable. For example, a different temperature probe might be used for another run in the same experiment. If one program is to be used to graph data passed through a variety of transforms, it is appropriate to place the transform in a defined function. The following is an example of this.

```
    .
    .
    .
110 INIT
115 DEF FNA(A)=5*(A-32)/9
120 VIEWPORT 20,120,20,80
    .
    .
    .
200 REM  CONVERT INPUT DATA FROM FAHRENHEIT TO CELSIUS
210 C(I)=FNA(F)
220 M1=M1 MIN C(I)
    .
    .
    .
```

The above example is identical to the one before it except that the transform used is defined in a function (in statement 115). Each subsequent use of the transform will be identical — merely a function call as in statement 210. So by changing one statement, the transform used by the entire program can be changed. The next section will illustrate a particularly useful application for this capability.

There are several classes of transformations which are commonly used for transforming data. (In the following formulae: X,Y,R, and T represent variables; A and B represent constants.)

TRANSFORMATION	EXAMPLE OF USAGE
Log to the base 10	DRAW X,A*LGT(B*Y)
Log to the base e	DRAW X,A*LOG(B*Y)
Polar (R = radius, T = angle)	DRAW R*COS(T),R*SIN(T)
Exponential	DRAW X,A ↑ (Y+B)
Power	DRAW X,Y↑A
Logit	DRAW LOG(X/(1−X)),Y for $0 \leqslant X \leqslant 1$

Section 5

AXIS

AXIS COMMAND REVIEW

An axis or grid makes graphed data much easier to understand. This section describes two classes of ways to add axes and grids to a graph: with use of the AXIS command, and without the use of AXIS. For most graphs, use of AXIS will be appropriate. Techniques which use the AXIS command are discussed first. However, there are some types of data for which use of AXIS is inappropriate, such as logarithmic and polar data. So the other class of procedures discussed is how to make axes and grids without using AXIS.

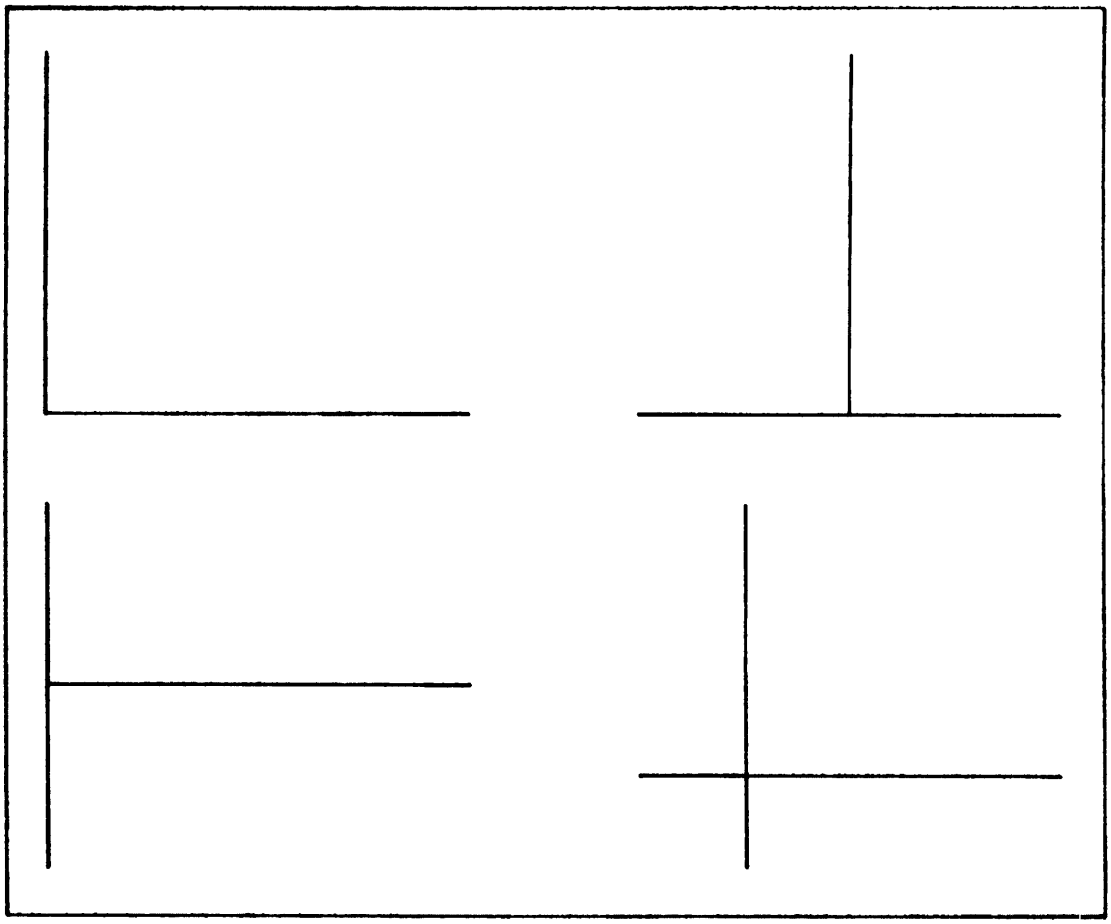
Without Arguments

The simplest way to add axis lines to a graph is with the AXIS command, described in Section 1. Here is a brief review. Please enter and run the following program:

```
100 PAGE
110 INIT
120 VIEWPORT 5,55,55,95
130 AXIS
140 REM
150 VIEWPORT 75,125,55,95
160 WINDOW -100,100,50,100
170 AXIS
180 REM
190 VIEWPORT 5,55,5,45
200 WINDOW 50,100,-100,100
210 AXIS
220 REM
230 VIEWPORT 75,125,5,45
240 WINDOW -25,75,-25,75
250 AXIS
260 HOME
270 END
```

Shown below is the output produced by the above program:

AXIS COMMAND REVIEW



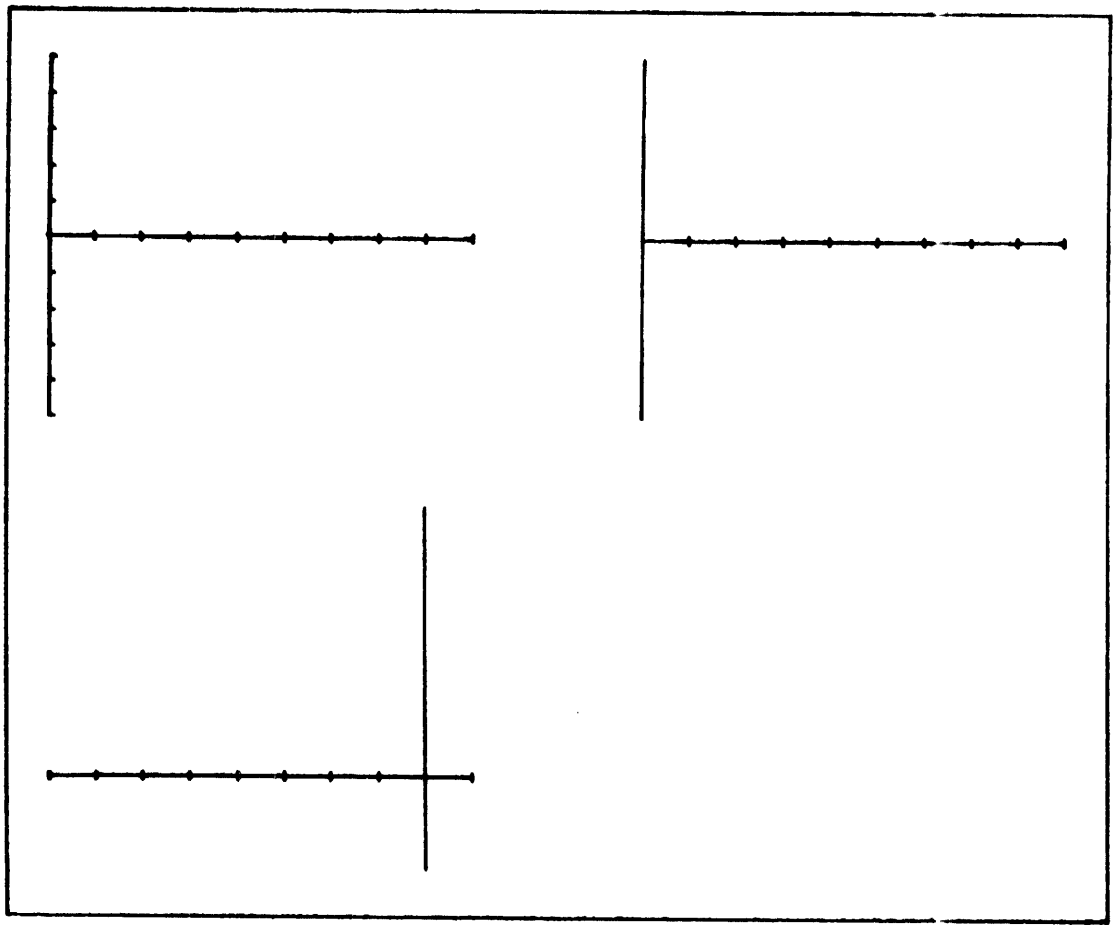
In the above program, the AXIS command is used four times. Each time it is used, no arguments are specified. When an AXIS command is executed with no arguments, the locations of the resulting axis lines are determined by the last defined window. With no arguments, AXIS draws each axis line through the data minimums specified in the last WINDOW command. However, if zero lies within either data range, the corresponding axis line is drawn through zero instead of the minimum. The above example program is a demonstration of how WINDOW determines the default locations of the AXIS lines.

When AXIS is first used (statement 130), only the default window (0,130,0,100) is defined. (The VIEWPORT command at statement 120 causes the window which contains the axis lines to be placed in the upper left part of the display.) In the second use of AXIS in the above example (the upper right set of axis lines in the above output), the WINDOW command at 160 places zero in the middle of the horizontal range. In the third use of AXIS (the lower left axis lines in the above output), zero is in the middle of the vertical data range. In the fourth use of AXIS, zero is near the lower end of both data ranges.

With Arguments

AXIS can also be used with two or four arguments. In either of those cases, the two arguments immediately following the AXIS command specify the distance in user data units between tic marks. The first argument refers to the horizontal axis tic marks and the second argument refers to the vertical axis tic marks. The following program illustrates how these arguments change the AXIS command's function.

```
100 PAGE
110 VIEWPORT 5,55,55,95
120 WINDOW 10,100,-100,100
130 AXIS 10,20
140 VIEWPORT 75,125,55,95
150 AXIS 10,0
160 VIEWPORT 5,55,5,45
170 AXIS 10,0,90,-50
180 HOME
190 END
```



Both axis lines in the upper left of the display (drawn by the AXIS command at statement 130) have tic marks. Tic marks are not drawn on an axis line if the corresponding argument in the AXIS command is zero. In statement 150, the second argument of the AXIS command is zero. As a result, the vertical axis in the upper right set of axes lines does not have tic marks.

The third and fourth arguments of the AXIS command specify the crossing point of the axes. The third argument specifies the horizontal location of the vertical axis. The fourth argument specifies the vertical location of the horizontal axis. In spite of their being drawn in an identical window, the set of axis lines in the lower left of the display cross at a different location than the other two sets of axis lines shown. The crossing point was specified by the third and fourth arguments of the AXIS command in statement 170.

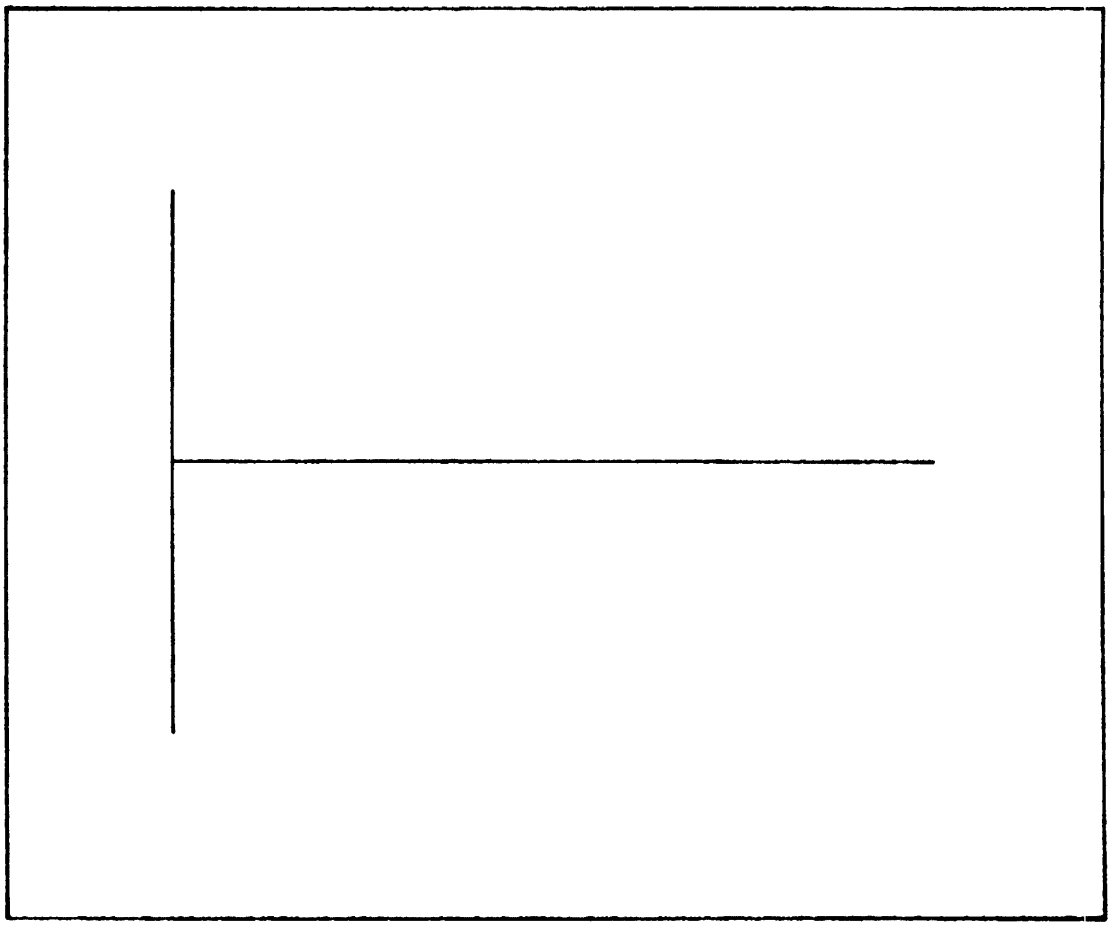
WITHOUT AXIS COMMAND

Another way to place an axis on the screen is to actually draw the lines using MOVE and DRAW. The remainder of this section describes how this is done for linear, logarithmic, and polar axis types. Understanding the material in the remainder of this section is necessary only for an application which requires an axis different from that produced by the AXIS command. If the axis requirements are adequately satisfied by the AXIS command, you may want to skip to the beginning of the next section.

Without Tic Marks

The next example uses MOVE and DRAW to place axis lines on the screen. For the vertical axis line, all this entails is moving to the point where the vertical axis and the Y data minimum meet (X and −500 in this case) and drawing to the point where the vertical axis and the Y data maximum meet. A similar process is used for the horizontal axis. Inherent in a program of this nature is the location for the crossing point of the two axis lines, which is X = 1950 and Y = 0 in the next example.

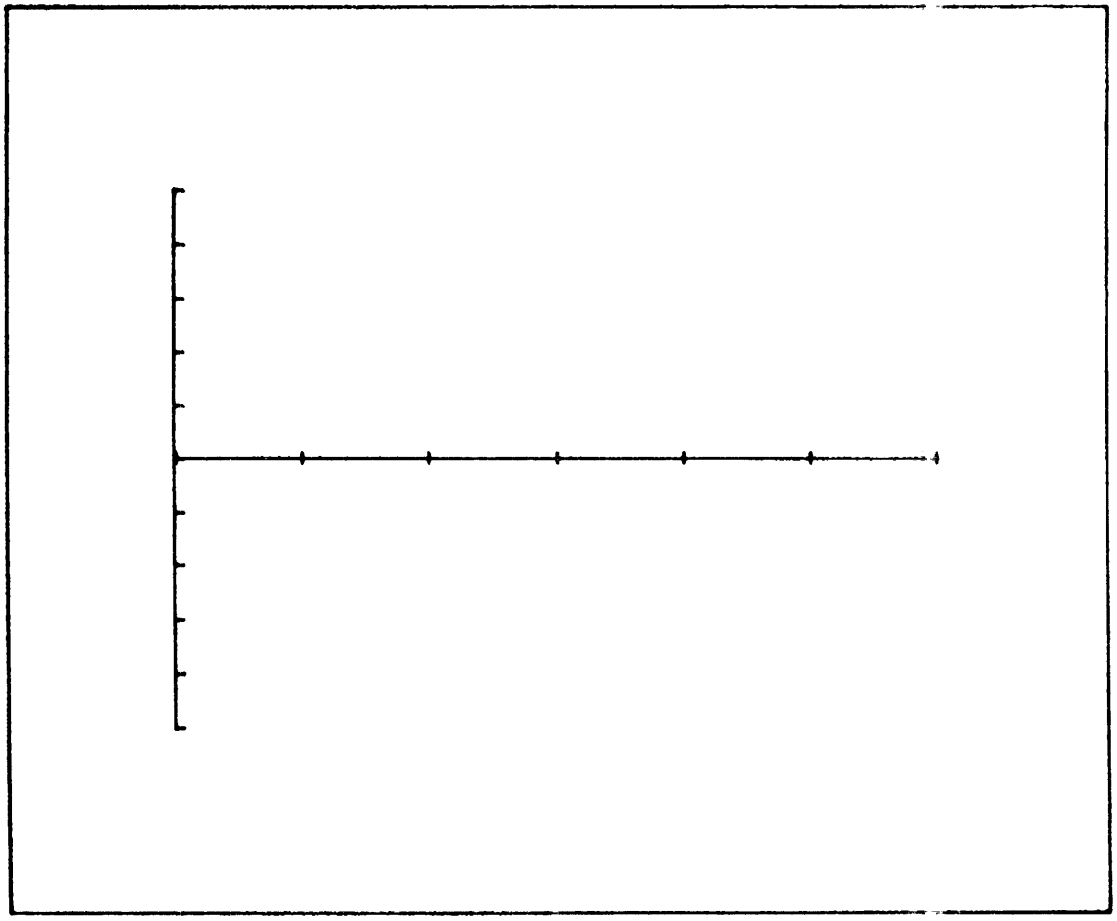
```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW 1950,1980,-500,500
130 REM  VERTICAL AXIS LINE
140 MOVE 1950,-500
150 DRAW 1950,500
160 REM  HORIZONTAL AXIS LINE
170 MOVE 1950,0
180 DRAW 1980,0
190 HOME
200 END
```



With Tic Marks

The addition of tic marks makes the axis lines more helpful in understanding the data graph. The best way to add tic marks is to use a MOVE and a DRAW in a FOR . . . NEXT loop. This is shown in the next example.

```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW 1950,1980,-500,500
130 REM  VERTICAL AXIS LINE
140 MOVE 1950,-500
150 DRAW 1950,500
160 REM  HORIZONTAL AXIS LINE
170 MOVE 1950,0
180 DRAW 1980,0
190 REM  VERTICAL AXIS TICS
200 REM   TIC LENGTH = 2% OF RANGE
210 T=(1980-1950)*0.01
220 FOR I=-500 TO 500 STEP 100
230 MOVE 1950-T,I
240 DRAW 1950+T,I
250 NEXT I
260 REM  HORIZONTAL AXIS TICS
270 T=(500--500)*0.01
280 FOR I=1950 TO 1980 STEP 5
290 MOVE I,-T
300 DRAW I,T
310 NEXT I
320 HOME
330 END
```



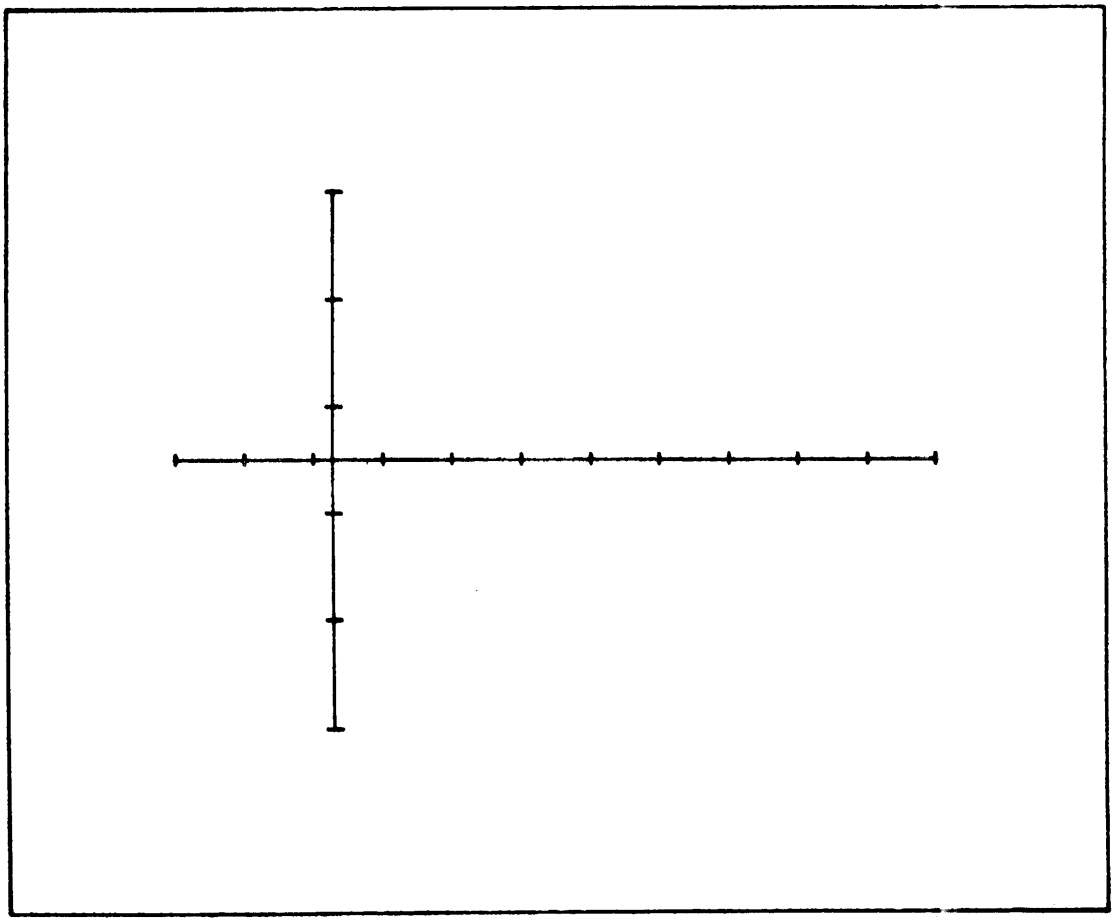
The axis crossing point and the distance between tic marks must be specified. With this technique, the distance between tic marks is implied by the STEP size in each FOR statement.

With the AXIS command, the actual size of the tic mark is defined to be 1% of the window size. With the technique in the above example, it must be explicitly specified. In the above example, it is 2%. (The length of the tics on the horizontal axis is 2% of the vertical data range.) The variable T contains a value equal to 1%: half of a horizontal axis tic is below the horizontal axis and half is above it. The total length is then 2%. This value can be made any appropriate length by changing statements 210 and 270. In the above example, the tics on the vertical axis are partially clipped because each tic's left half is outside the window.

Unaligned Tic Marks

Here is the same program with slightly different data.

```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW -23,87,-25,25
130 REM  VERTICAL AXIS LINE
140 MOVE 0,-25
150 DRAW 0,25
160 REM  HORIZONTAL AXIS LINE
170 MOVE -23,0
180 DRAW 87,0
190 REM  VERTICAL AXIS TICS
200 REM   TIC LENGTH = 2% OF RANGE
210 T=(87--23)*0.01
220 FOR I=-25 TO 25 STEP 10
230 MOVE 0-T,I
240 DRAW 0+T,I
250 NEXT I
260 REM  HORIZONTAL AXIS TICS
270 T=(25--25)*0.01
280 FOR I=-23 TO 87 STEP 10
290 MOVE I,-T
300 DRAW I,T
310 NEXT I
320 HOME
330 END
```



In the above example, the tics are aligned with the window, not the axis intersection point. This is caused by each FOR . . . NEXT loop using the data minimum for a starting point. For example, the FOR statement at line 220 specifies a starting value of −25, the vertical data minimum. This will always produce a tic mark on the lower end of the vertical axis, at the bottom edge of the window.

Alignment Correction

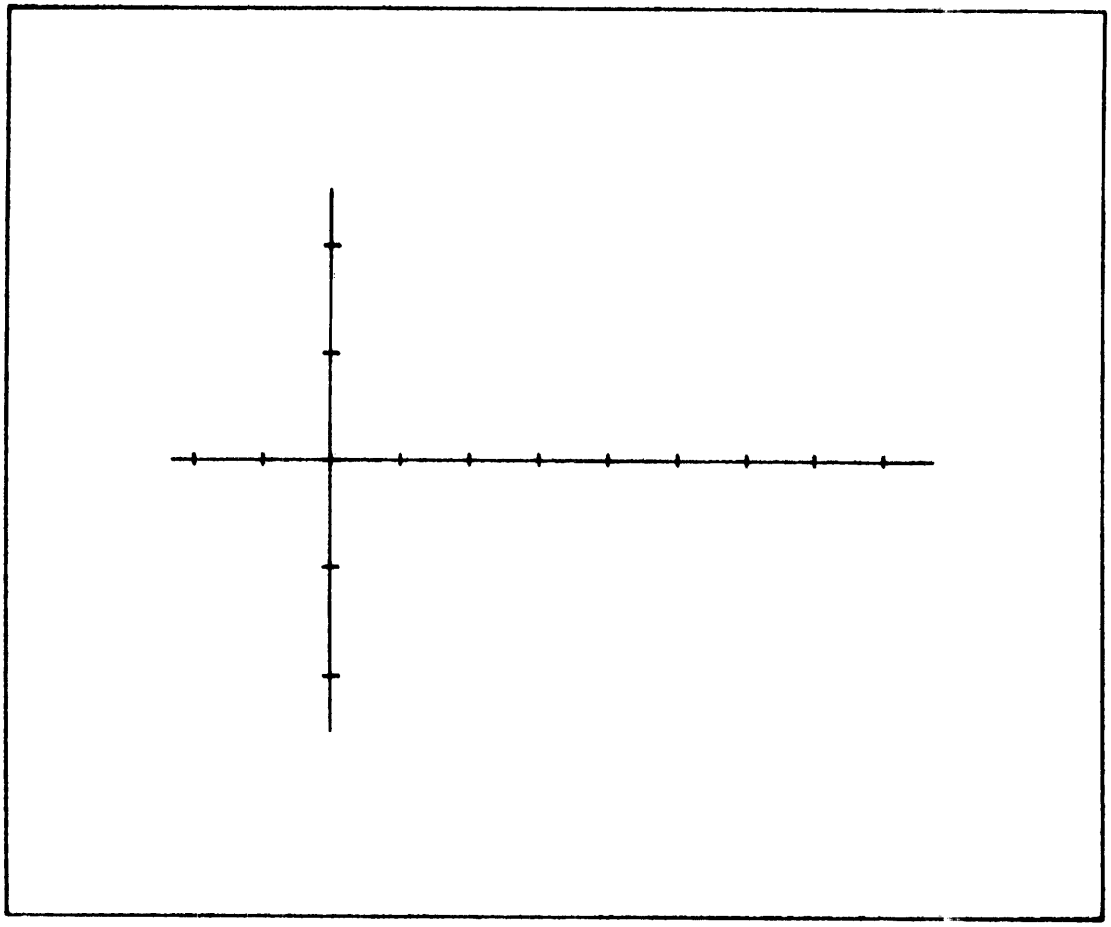
If the tics are always to be aligned with the axis crossing point, the following correction can be used:

```
    .
    .
    .
215 U=(25/10-INT(25/10))*10
220 FOR I=-25+U TO 25 STEP 10
    .
    .
    .
275 U=(23/10-INT(23/10))*10
280 FOR I=-23+U TO 87 STEP 10
    .
    .
    .
```

(Statements 215 and 275 are added; statements 220 and 280 are changed.)

This correction changes the starting point of each FOR . . . NEXT loop. The change is just enough to place a tic mark at the intersection point of each axis line. A more complete explanation of the correction follows the next example. Here is what the complete program looks like:

```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW -23,87,-25,25
130 REM  VERTICAL AXIS LINE
140 MOVE 0,-25
150 DRAW 0,25
160 REM  HORIZONTAL AXIS LINE
170 MOVE -23,0
180 DRAW 87,0
190 REM  VERTICAL AXIS TICS
200 REM   TIC LENGTH = 2% OF RANGE
210 T=(87--23)*0.01
215 U=(25/10-INT(25/10))*10
220 FOR I=-25+U TO 25 STEP 10
230 MOVE 0-T,I
240 DRAW 0+T,I
250 NEXT I
260 REM  HORIZONTAL AXIS TICS
270 T=(25--25)*0.01
275 U=(23/10-INT(23/10))*10
280 FOR I=-23+U TO 87 STEP 10
290 MOVE I,-T
300 DRAW I,T
310 NEXT I
320 HOME
330 END
```



In each FOR . . . NEXT loop, the value placed in U is the distance between the edge of the window and the first tic mark.

General Correction

To show how this correction is applicable to the general case, the next example replaces the number constants with variables. Here is a list of the variables used:

M1 = horizontal data minimum
M2 = horizontal data maximum
N1 = vertical data minimum
N2 = vertical data maximum
X = horizontal location of vertical axis
Y = vertical location of horizontal axis (X,Y is the crossing point of the two axis lines)
I1 = distance between tics, horizontal axis
I2 = distance between tics, vertical axis

```
100 PAGE
110 VIEWPORT 20,110,20,80
113 RESTORE
115 DATA -23,87,-25,25,0,0,10,10
117 READ M1,M2,N1,N2,X,Y,I1,I2
120 WINDOW M1,M2,N1,N2
130 REM  VERTICAL AXIS LINE
140 MOVE X,N1
150 DRAW X,N2
160 REM  HORIZONTAL AXIS LINE
170 MOVE M1,Y
180 DRAW M2,Y
190 REM  VERTICAL AXIS TICS
200 REM   TIC LENGTH = 2% OF RANGE
210 T=(M2-M1)*0.01
215 U=((Y-N1)/I2-INT((Y-N1)/I2))*I2
220 FOR I=N1+U TO N2 STEP I2
230 MOVE X-T,I
240 DRAW X+T,I
250 NEXT I
260 REM  HORIZONTAL AXIS TICS
270 T=(N2-N1)*0.01
275 U=((X-M1)/I1-INT((X-M1)/I1))*I1
280 FOR I=M1+U TO M2 STEP I1
290 MOVE I,Y-T
300 DRAW I,Y+T
310 NEXT I
320 HOME
330 END
```

(Since the output from this program is identical to the output from the one before it, no output is shown.)

As stated before, the corrections included at statements 215 and 275 merely find the distance between the edge of the window and the first tic. For the case of the vertical axis correction (line 215), here is a detailed description of the process:

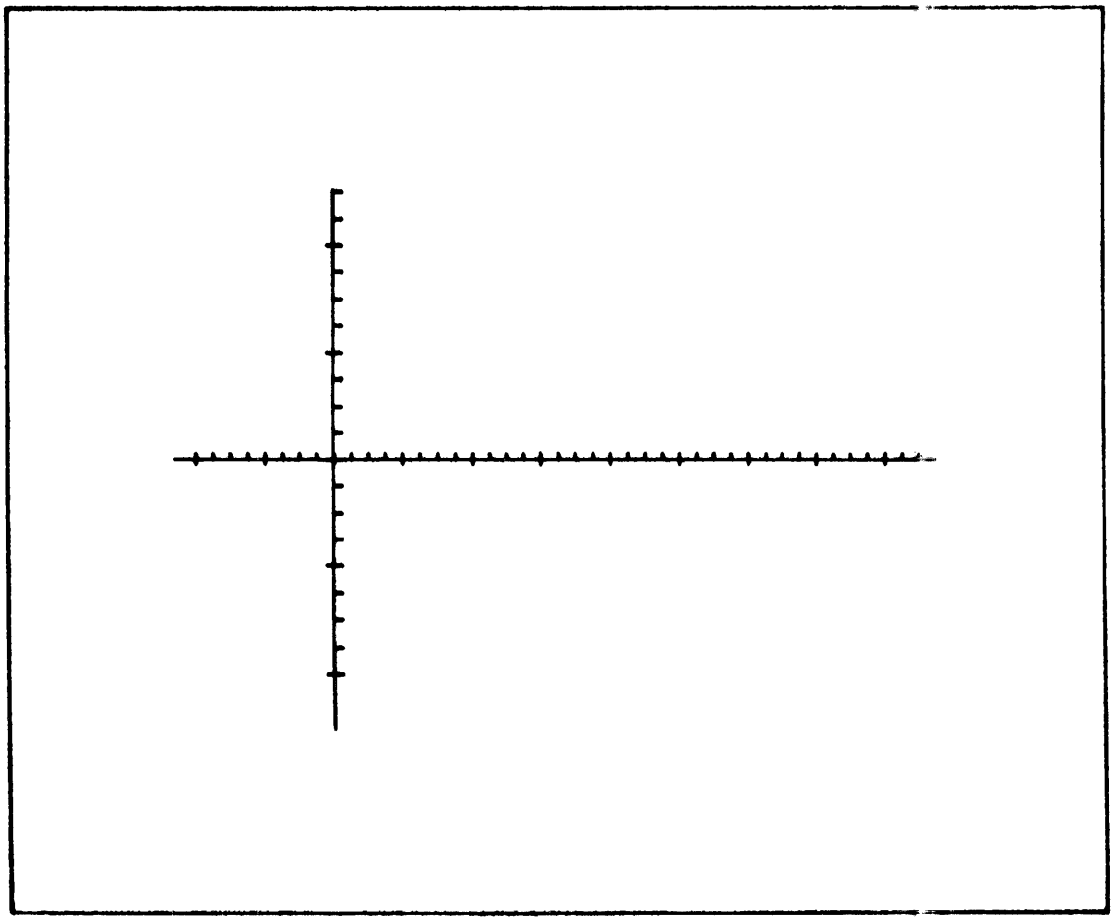
1. Find the distance between the axis crossing and the data minimum (in this example it is Y—N1).
2. Divide this distance by the distance between tics (in this case I 2). This quotient is the total number of full and partial tic intervals which will fit between the vertical data minimum and the axis crossing point.
3. Subtract from this number its integer part. What remains is the distance between the edge of the window and the first tic, expressed as a decimal fraction of a tic interval.
4. Multiply this fraction by the distance between tics. The product is the desired distance.

This program duplicates the function of the AXIS command, a capability which will be useful when labeling tic marks.

Minor Tic Marks

The addition of minor tic marks, i.e., marks that are shorter than the major tic marks, can often make a graph clearer. If the following statements are added to it, the example program above will draw minor tic marks. (There are four minor tic intervals for each major tic interval.)

```
      .
      .
      .
242 FOR J=I TO I+I2 STEP I2/4
244 MOVE X,J
246 DRAW X+T,J
248 NEXT J
      .
      .
      .
302 FOR J=I TO I+I1 STEP I1/4
304 MOVE J,Y
306 DRAW J,Y+T
308 NEXT J
      .
      .
      .
```



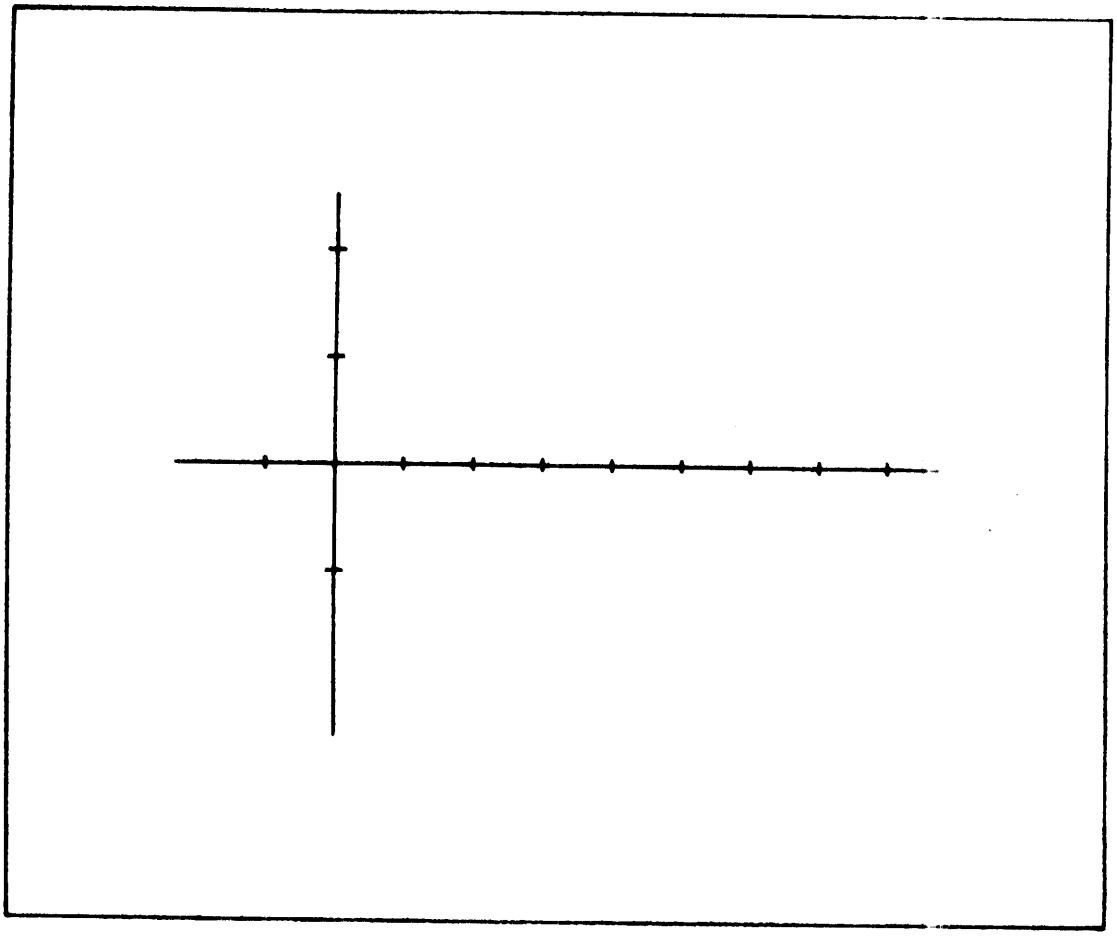
Only statements 242 through 248 and 302 through 308 were added. For the vertical axis, statements 242 through 248 comprise an inner loop which draws the minor tics in the same manner that the outer loop (statements 220 through 250) draw the major tics. Statements 302 through 308 serve a similar function for the horizontal axis.

Using RDRAW

Another way to draw axis lines with major tic marks is to use the relative RDRAW command instead of the absolute DRAW. All the examples so far have used DRAW. There will be a few instances where the use of RDRAW and RMOVE is appropriate for drawing an axis. However, in most cases, using MOVE and DRAW will produce a program which occupies slightly less space in machine memory and which executes faster. The example below draws axis lines and tic marks using RDRAW:

```
100 PAGE
110 VIEWPORT 20,110,20,80
113 RESTORE
115 DATA -23,87,-25,25,0,0,10,10
117 READ M1,M2,N1,N2,X,Y,I1,I2
120 WINDOW M1,M2,N1,N2
130 REM  VERTICAL AXIS
140 T=(M2-M1)*0.01
150 U=((Y-N1)/I2-INT((Y-N1)/I2))*I2
160 MOVE X,N1
170 RDRAW 0,U
180 FOR I=U+N1 TO N2 STEP I2
190 RDRAW 0,I2
200 RDRAW T,0
210 RDRAW -T*2,0
220 RDRAW T,0
230 NEXT I
240 REM  HORIZONTAL AXIS
250 T=(N2-N1)*0.01
260 U=((X-M1)/I1-INT((X-M1)/I1))*I1
270 MOVE M1,Y
280 RDRAW U,0
290 FOR I=U+M1 TO M2 STEP I1
300 RDRAW I1,0
310 RDRAW 0,T
320 RDRAW 0,-T*2
330 RDRAW 0,T
340 NEXT I
350 HOME
360 END
```

WITHOUT AXIS COMMAND

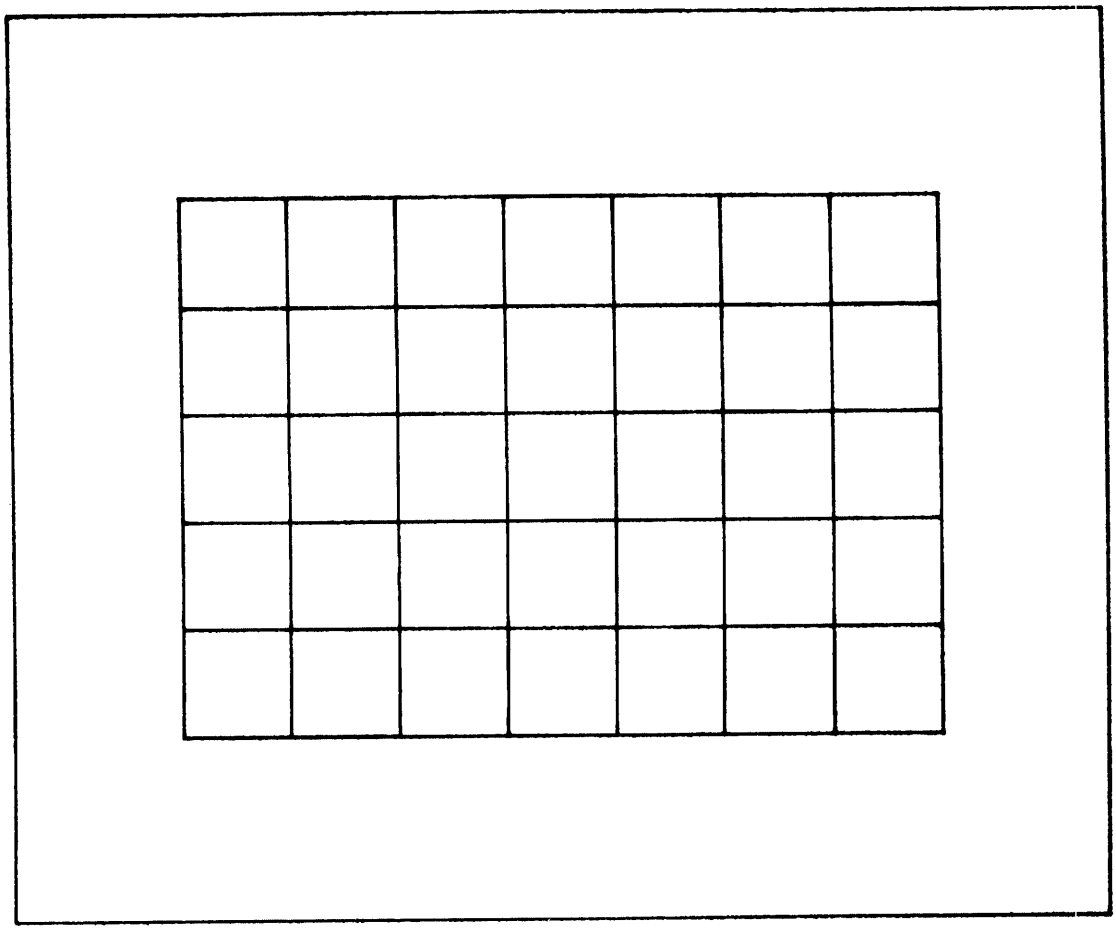


GRIDS

Lines

The next example shows that a grid covering the whole window can be drawn by repeated use of the AXIS command within a FOR . . . NEXT loop.

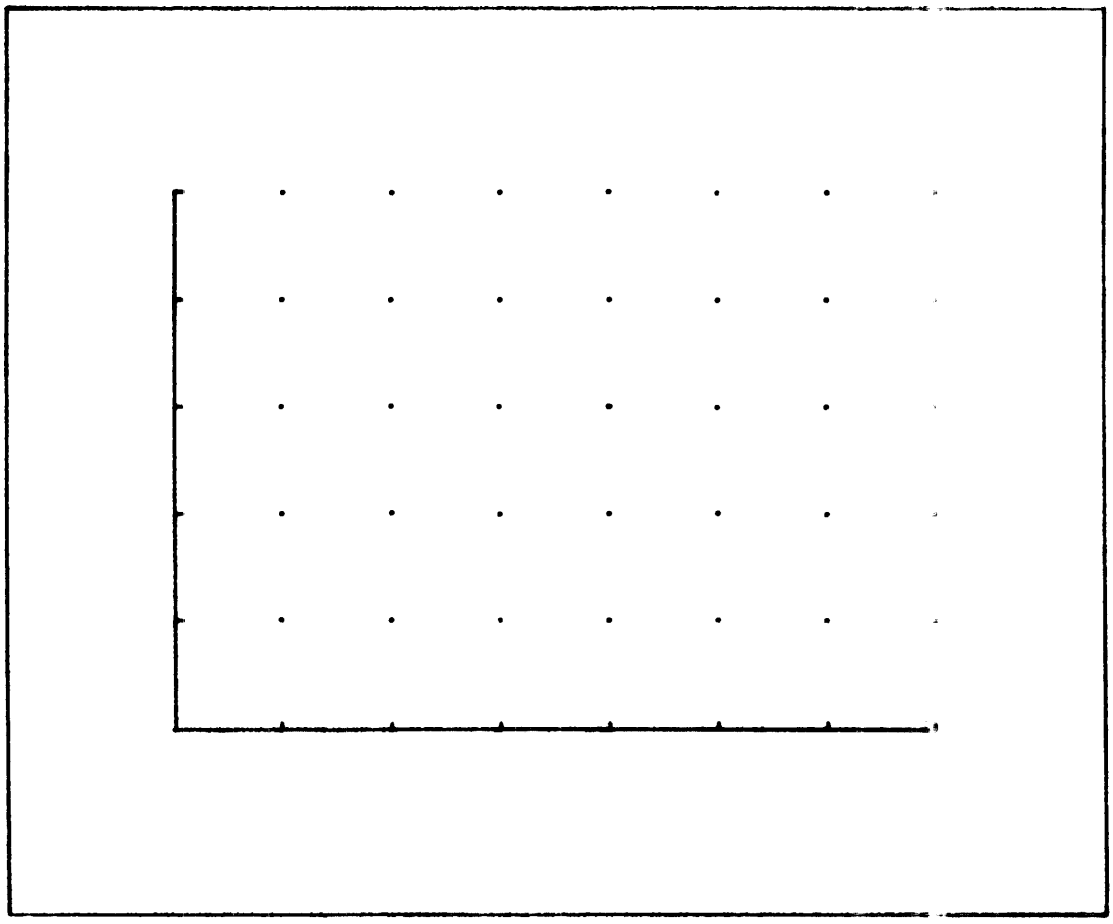
```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW 0,140,0,100
130 FOR I=0 TO 140 STEP 20
140 AXIS 0,0,I,I
150 NEXT I
160 HOME
170 END
```



Dots

For some applications, a grid such as the one shown above clutters the window excessively. A grid of dots, shown in the example below, is sometimes more satisfactory.

```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW 0,140,0,100
125 AXIS 20,20
130 FOR I=20 TO 140 STEP 20
140 FOR J=20 TO 100 STEP 20
150 MOVE I,J
160 DRAW I,J
170 NEXT J
180 NEXT I
190 HOME
200 END
```

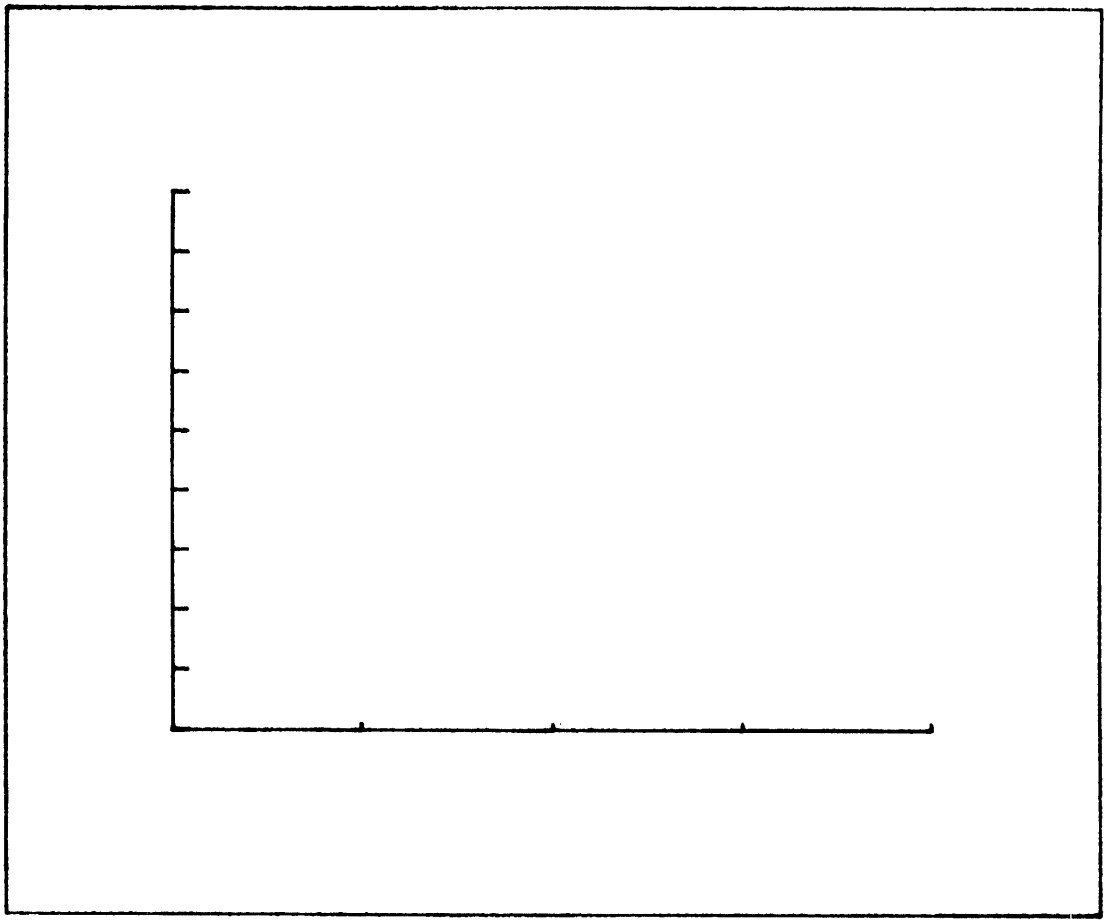


LOG AXIS

Linear Example

Non-linear axis types, appropriate for certain transformations, can be drawn using the techniques of previous examples. The next example shows a vertical axis with tic marks drawn at linear intervals.

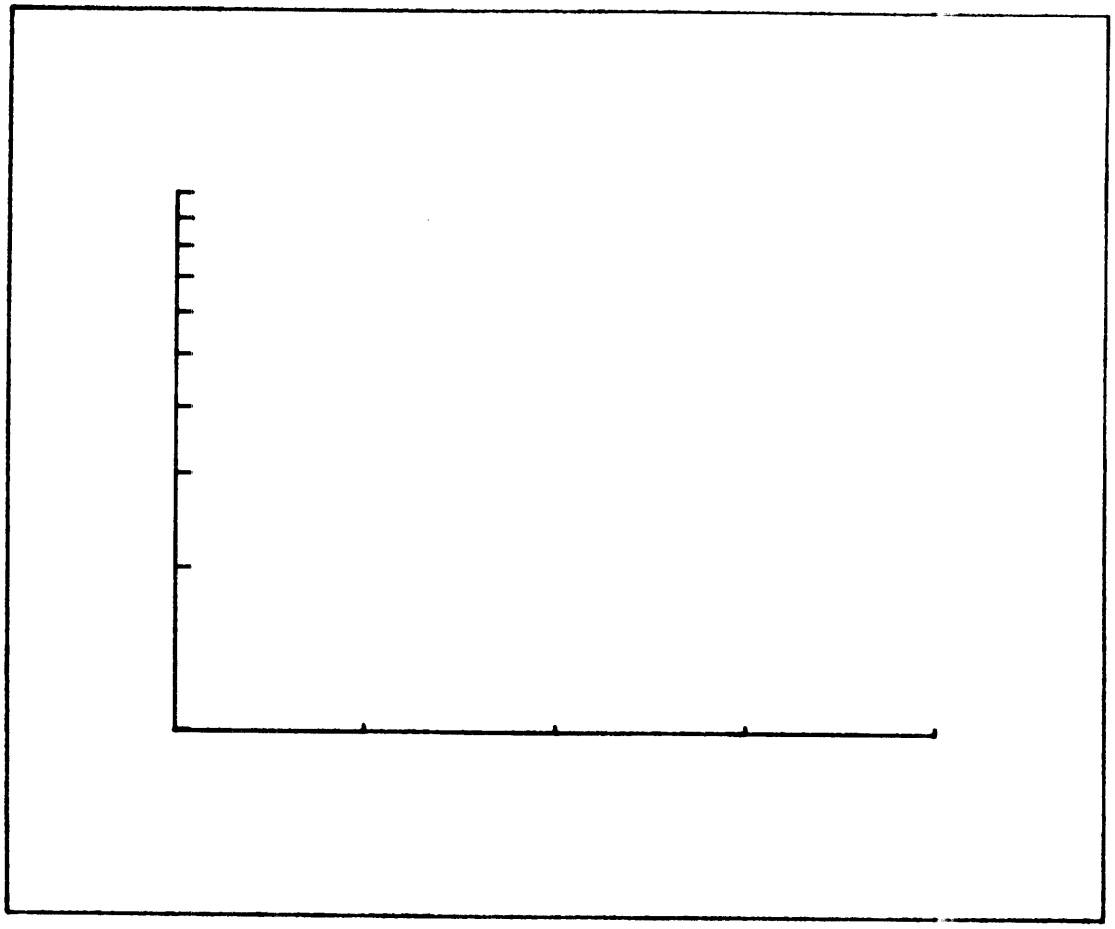
```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW 1,5,1,10
130 REM  DRAW AXIS LINES
140 AXIS 1,0,1,1
150 REM  DRAW VERTICAL AXIS TICS WITH LINEAR INTERVAL
160 REM   TIC MARK LENGTH IS 2% OF DATA RANGE
170 T=(5-1)*0.02
180 FOR I=1 TO 10
190 MOVE 1,I
200 DRAW 1+T,I
210 NEXT I
220 HOME
230 END
```



Adapted to Log

An often used non-linear transformation is log to the base 10. In the example below, the tic marks on the vertical axis are drawn with a log interval between them. The example below is adapted from the example above with only a few modifications.

```
100 PAGE
110 VIEWPORT 20,110,20,80
120 WINDOW 1,5,LGT(1),LGT(10)
130 REM  DRAW AXIS LINES
140 AXIS 1,0
150 REM  DRAW VERTICAL AXIS TICS WITH LOG INTERVAL
160 REM   TIC MARK LENGTH IS 2% OF DATA RANGE
170 T=(5-1)*0.02
180 FOR I=1 TO 10
190 MOVE 1,LGT(I)
200 DRAW 1+T,LGT(I)
210 NEXT I
220 HOME
230 END
```



Just as in the previous example, statement 140 draws the axis lines and statements 170 through 210 draw the tic marks. The only changes required were in the WINDOW, MOVE and DRAW commands (statements 120,190, and 200, respectively). In those commands, each reference to vertical data has been made the argument of the LGT function. These are the only changes required.

Less Than One Decade

The previous example's data range runs from 1 to 10, an exact decade. Quite often it is necessary to graph data with a range of less than one decade. The simplest way to draw a log axis for such data is to expand the graphed data range so that it will span exactly one decade. (The data range of the original data remains unchanged.) The example below illustrates this technique.

N1 = Actual vertical data minimum

N2 = Actual vertical data maximum

N3 = Adjusted vertical data minimum

N4 = Adjusted vertical data maximum

(The output of the program below is identical to the axis presented immediately above. For this reason, it is not shown.)

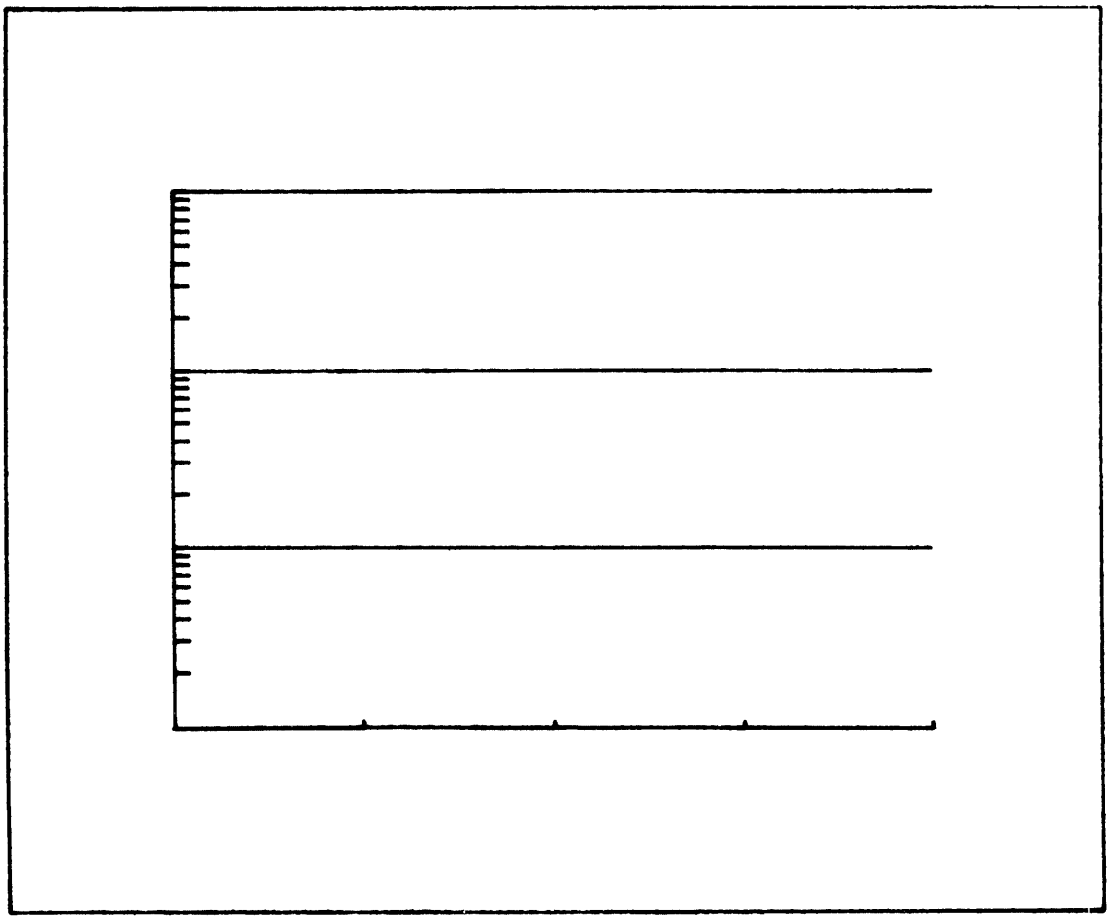
```
100 PAGE
110 VIEWPORT 20,110,20,80
111 N1=2000
112 N2=8000
113 N3=10↑INT(LGT(N1))
114 N4=10↑(INT(LGT(N2))+1)
115 IF N4/N3<=10 THEN 120
116 PRINT "ERROR:  N4 > 10*N3"
117 END
120 WINDOW 1,5,LGT(N3),LGT(N4)
130 REM  DRAW AXIS LINES
140 AXIS 1,0
150 REM  DRAW VERTICAL AXIS TICS WITH LOG INTERVAL
160 REM   TIC MARK LENGTH IS 2% OF DATA RANGE
170 T=(5-1)*0.02
180 FOR I=N3 TO N4 STEP N3
190 MOVE 1,LGT(I)
200 DRAW 1+T,LGT(I)
210 NEXT I
220 HOME
230 END
```

The only additional statements required are 111 through 117. Statements 111 and 112 specify the actual data minimum and maximum. Statements 113 and 114 adjust the range. Statement 115 ensures that the data is within one decade. Statements 116 and 117 comprise an error exit.

More Than One Decade

The example below handles more than one decade with a similar adjustment technique:

```
100 PAGE
110 VIEWPORT 20,110,20,80
111 N1=2000
112 N2=380000
113 N3=10↑INT(LGT(N1))
114 N4=10↑(INT(LGT(N2))+1)
120 WINDOW 1,5,LGT(N3),LGT(N4)
130 REM  DRAW AXIS LINES
140 AXIS 1,0
150 REM  DRAW VERTICAL AXIS TICS WITH LOG INTERVAL
160 REM   TIC MARK LENGTH IS 2% OF DATA RANGE
170 T=(5-1)*0.02
175 H=LGT(N3)
180 FOR I=10↑H TO 10↑(H+1) STEP 10↑H
190 MOVE 1,LGT(I)
200 DRAW 1+T,LGT(I)
210 NEXT I
211 H=H+1
212 DRAW 5,LGT(10↑H)
213 IF H<LGT(N4) THEN 180
220 HOME
230 END
```



This example is identical to the one which preceded it with the following exceptions:

Statements 115 through 117 have been removed. They are no longer needed.

Statements 175 has been added. This sets up the beginning value for the first decade to be drawn.

Statement 180 has been changed. It still draws one decade's tic marks but its beginning point, ending point, and increment change during program execution.

Statement 212 has been added. This is an optional statement. It is really just a "major tic" which extends across the entire graphing surface. This adds clarity to the graphed results. If statement 212 is removed, a normal tic will remain.

Statement 213 checks if the axis has been completely drawn, and branches back into the loop if it has not.

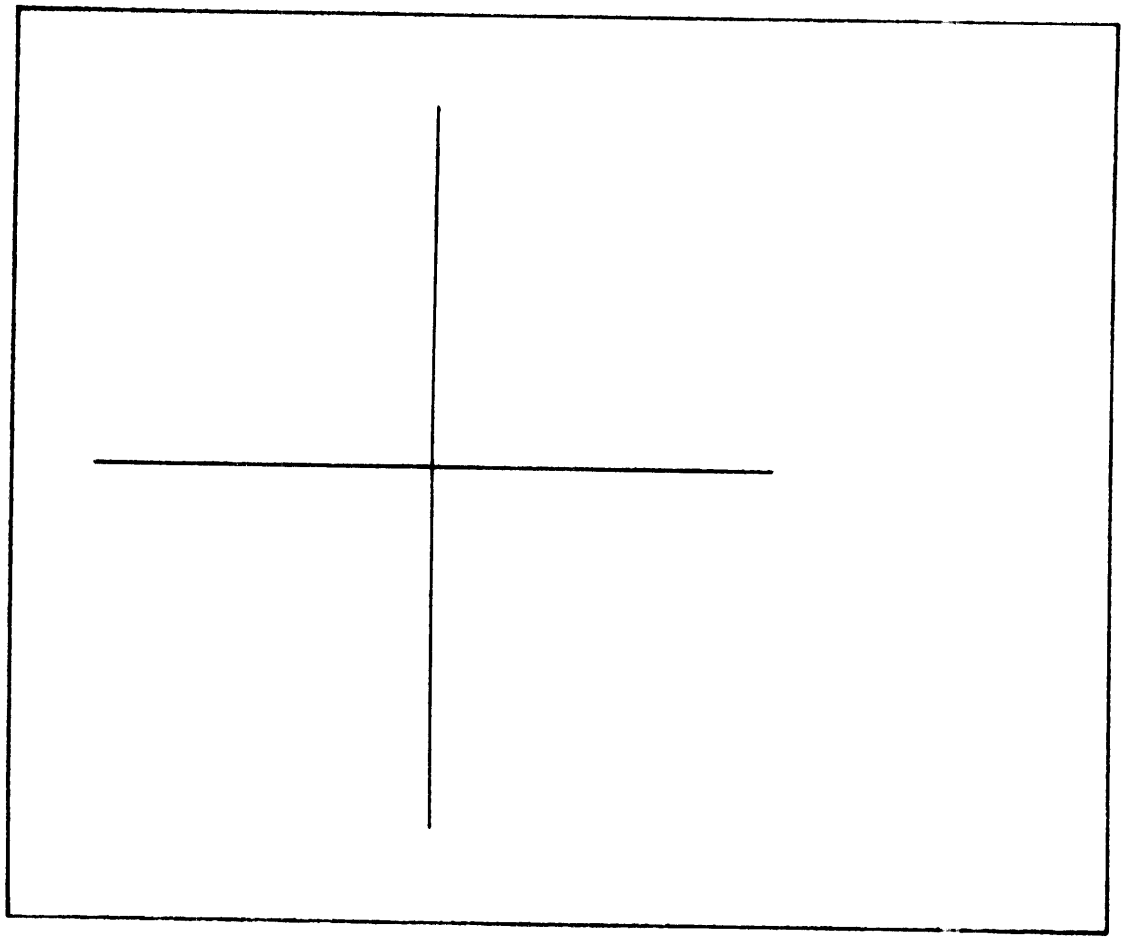
Statement 211 updates the starting point for the loop which draws the tic marks.

POLAR AXIS

Two Axis Lines

The polar transformation is also very useful. The simplest polar axis is just 2 crossed lines, dividing the graphing window into quadrants. The next example shows how a FOR . . . NEXT loop is used to draw these lines. The STEP value in the FOR . . . NEXT loop specifies the included angle between adjacent axis lines.

```
100 PAGE
110 SET DEGREES
120 REM  R1 = MAXIMUM RADIUS VALUE
130 R1=9
140 WINDOW -R1,R1,-R1,R1
150 VIEWPORT 10,90,10,90
160 REM  DRAW AXIS LINES
170 FOR I=0 TO 180 STEP 90
180 MOVE R1*COS(I),R1*SIN(I)
190 DRAW R1*COS(I+180),R1*SIN(I+180)
200 NEXT I
210 HOME
220 END
```

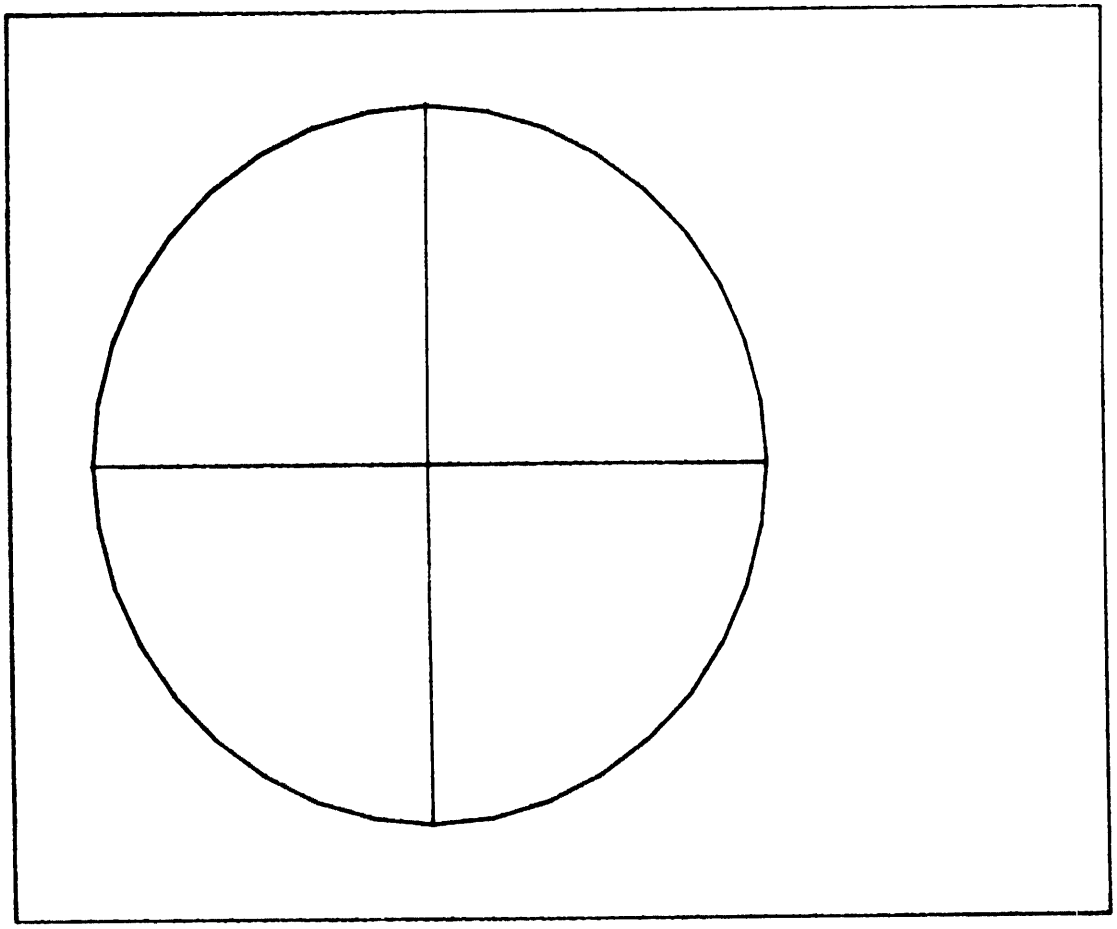


In the above example, the defined window and viewport both have the same shape (square). If they did not the data would be distorted, causing a circle to be drawn as an ellipse.

Two Lines and One Circle

The next example adds a subroutine which can draw a circle of any given radius.

```
100 PAGE
110 SET DEGREES
120 REM  R1 = MAXIMUM RADIUS VALUE
130 R1=9
140 WINDOW -R1,R1,-R1,R1
150 VIEWPORT 10,90,10,90
160 REM  DRAW AXIS LINES
170 FOR I=0 TO 180 STEP 90
180 MOVE R1*COS(I),R1*SIN(I)
190 DRAW R1*COS(I+180),R1*SIN(I+180)
200 NEXT I
210 REM  DRAW CIRCLE OF RADIUS  R
220 R=R1
230 GOSUB 500
240 HOME
250 END
500 REM  SUBROUTINE TO DRAW CIRCLE
510 MOVE R,0
520 FOR J=10 TO 360 STEP 10
530 DRAW R*COS(J),R*SIN(J)
540 NEXT J
550 RETURN
```

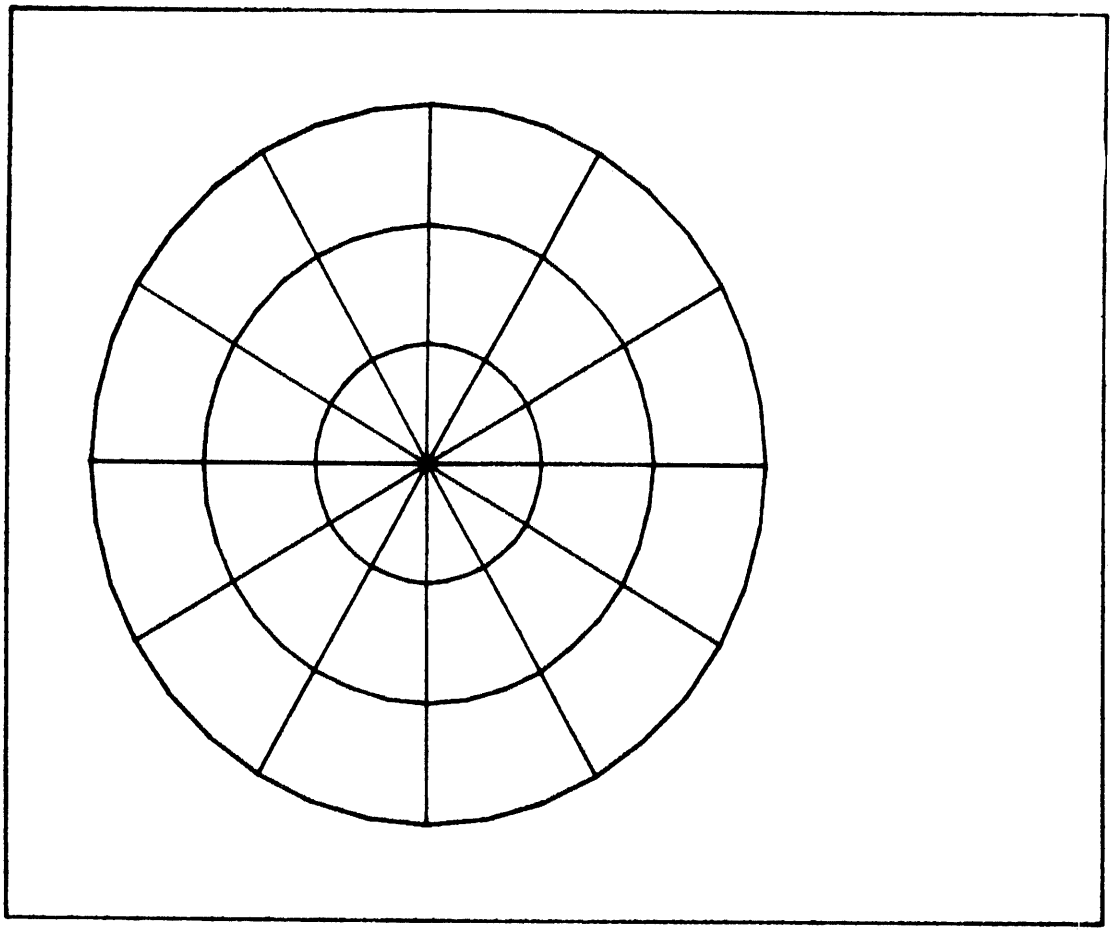


In the above example, the straight lines which comprise the circle are just becoming discernible. This is due to the STEP size in statement 520 being relatively large. A smaller value results in a somewhat more satisfactory-looking circle which takes longer to draw.

Multiple Lines and Circles

The example below shows the axis line loop and the circle subroutine being used to make additional lines and circles.

```
100 PAGE
110 SET DEGREES
120 REM  R1 = MAXIMUM RADIUS VALUE
130 R1=9
140 WINDOW -R1,R1,-R1,R1
150 VIEWPORT 10,90,10,90
160 REM  DRAW AXIS LINES
170 FOR I=0 TO 180 STEP 30
180 MOVE R1*COS(I),R1*SIN(I)
190 DRAW R1*COS(I+180),R1*SIN(I+180)
200 NEXT I
210 REM  DRAW CIRCLE OF RADIUS  R
220 R=R1
230 GOSUB 500
240 R=R1*(2/3)
250 GOSUB 500
260 R=R1/3
270 GOSUB 500
280 HOME
290 END
500 REM  SUBROUTINE TO DRAW CIRCLE
510 MOVE R,0
520 FOR J=10 TO 360 STEP 10
530 DRAW R*COS(J),R*SIN(J)
540 NEXT J
550 RETURN
```

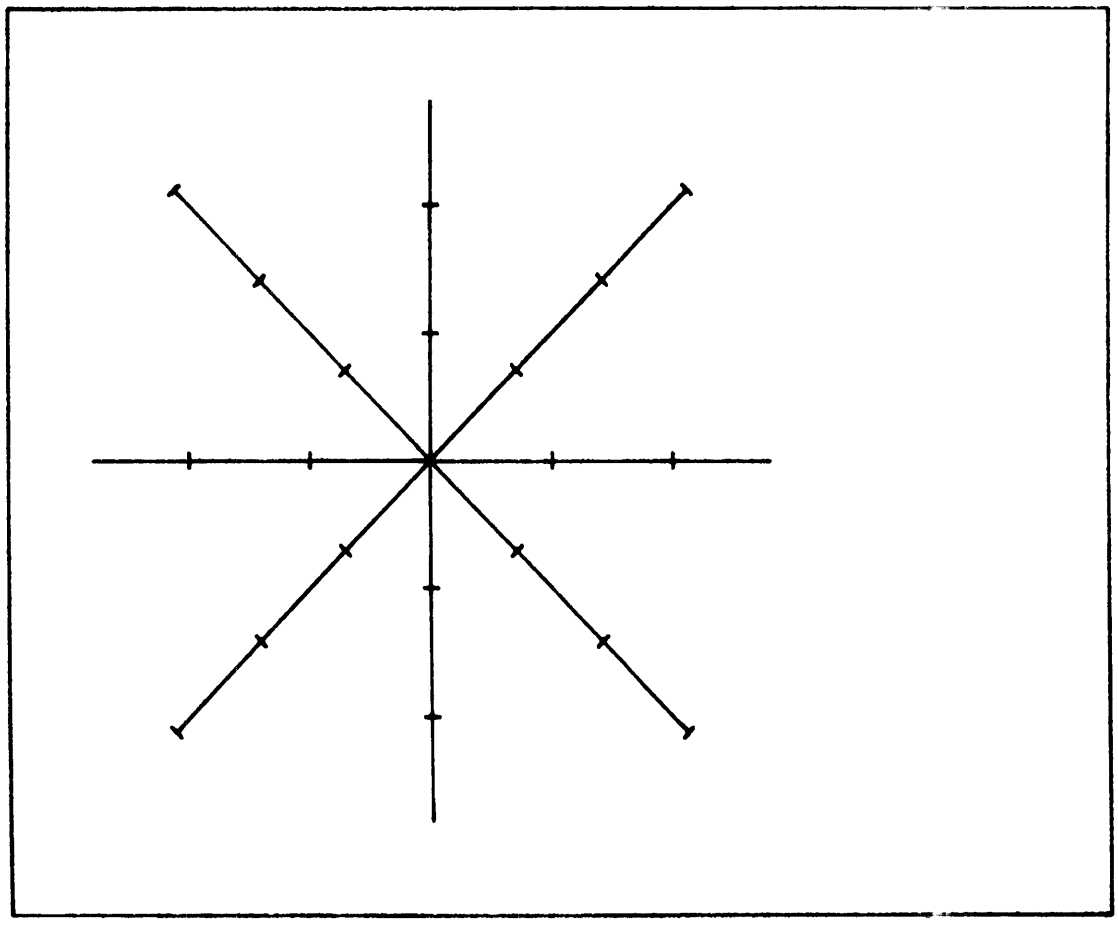


In the above example, the STEP value in the FOR . . . NEXT loop (beginning at statement 170) has been changed from 90 to 30. As a result, the axis lines now occur every 30 degrees. The circle subroutine has been used 3 times: with the maximum radius value, with 2/3 the maximum radius value and with 1/3 the maximum radius value. Some care should be exercised when choosing how many axes and circles to place on a polar graph. Too many lines and circles lead to a cluttering effect which can obscure data.

Using RDRAW

A slightly different approach will avoid this cluttering effect: a polar axis made up of just straight lines and tic marks. RDRAW is the most appropriate command for this purpose. The next example illustrates this.

```
100 PAGE
110 SET DEGREES
120 REM  R1 = MAXIMUM RADIUS VALUE
130 R1=14
140 WINDOW -R1,R1,-R1,R1
150 VIEWPORT 10,90,10,90
160 REM  A IS ANGLE BETWEEN AXIS LINES
170 A=45
180 REM  I IS TIC INTERVAL
190 I=5
200 REM  DRAW AXIS LINES
210 REM   T IS SIZE OF TIC MARKS
220 T=R1*0.02
230 FOR J=A TO 360 STEP A
240 ROTATE J
250 MOVE 0,0
260 FOR K=I TO R1+I STEP I
270 RDRAW I,0
280 RDRAW 0,T
290 RDRAW 0,-T*2
300 RDRAW 0,T
310 NEXT K
320 NEXT J
330 HOME
340 END
```



Two nested FOR . . . NEXT loops draw the axes in the above example. The inner loop (statements 260 through 310) draws each axis line with tic marks. The outer loop (statements 230 through 320) determines how many axis lines will be drawn.

It is desirable for the tic marks to be lined up with the pole of the graph. The easiest way to accomplish this is to start each axis line at the pole and draw it outward. This requires that the outer FOR . . . NEXT loop (beginning at statement 230) extend through one full revolution of 360 degrees rather than through one half revolution of 180 degrees, as in the above examples. The inner FOR . . . NEXT loop's terminating value (R1 + I in statement 260) ensures that the axis lines will extend outward at least as far as the maximum radius, R1. If the terminating value of the inner loop is specified (in statement 260) as simply R1, the axis lines will extend outward only as far as the last tic inside a radius of R1, an unsatisfactory condition. The axis lines should extend at least as far as R1 and preferably farther. There is no particular disadvantage in having the axis lines extend too far (other than excessive execution time). Any part of an axis line or tic mark which extends outside the defined window is clipped. This is why half the axis lines in the above example have terminating tic marks and half do not.

Section 6

LABELS

INTRODUCTION

There are three types of character information customarily added to a graph: the title, the axis label, and the tic mark labels.

There are literally dozens of ways to title and label a graph. This section discusses the considerations which influence the addition of characters to a graph and suggests examples of how the required tasks can be accomplished.

Adding characters to a graph in an orderly fashion is a three step process:

1. Making sure that there is appropriate space on the display for the characters.
2. Positioning the beginning point of the character output so that the characters will correctly fill the reserved space.
3. Printing the characters.

This entire section describes various ways of performing these three steps.

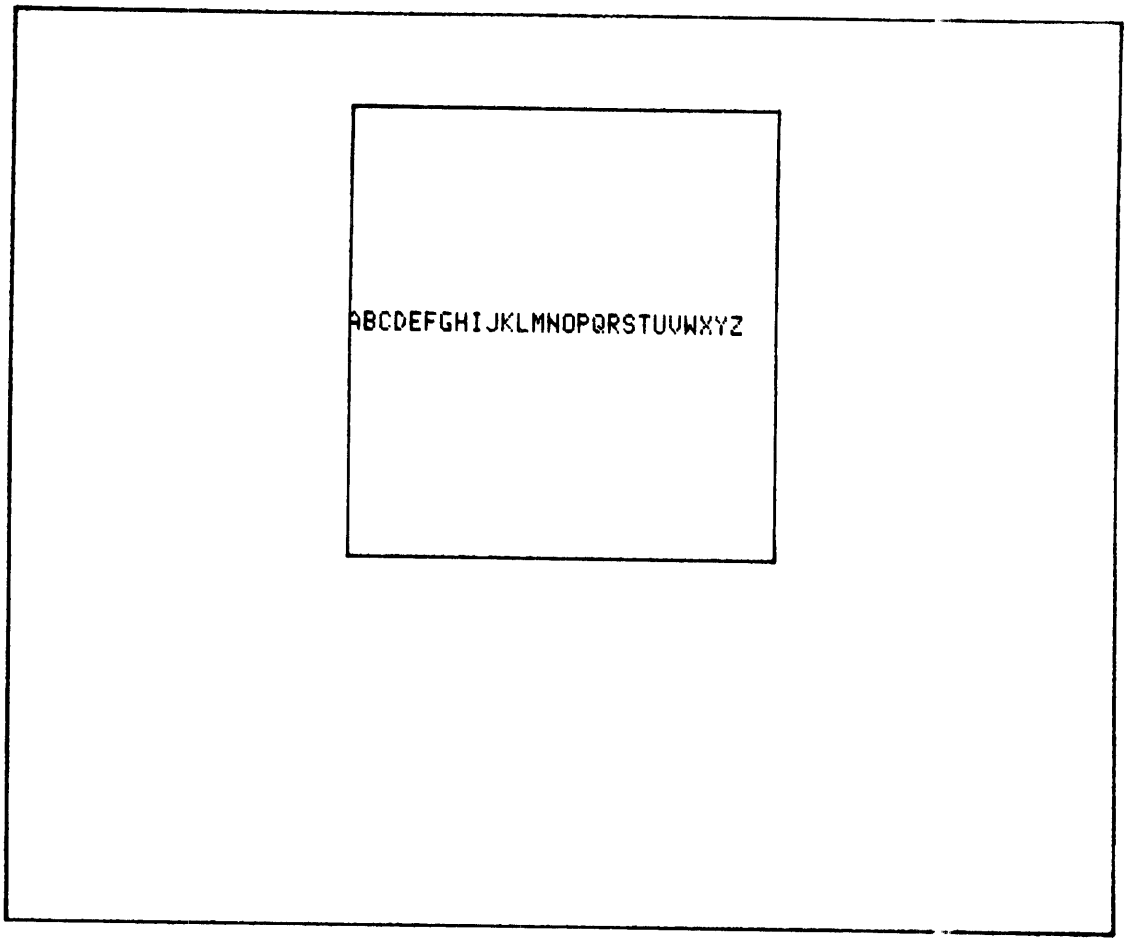
CONSTANT SIZE

The most significant consideration for labeling graphs on the GS display is that the size and orientation of the characters on the display are constant. For example, the size and orientation of a letter "A" is not affected by any previously executed commands. In contrast, the length and orientation of a line produced by a DRAW command is determined by the last executed WINDOW and VIEWPORT commands. The following examples show some of the implications of this fact.

The first example, demonstrates how the size of the viewport does not affect the size of the characters. Enter the following statements into the GS:

```
100 PAGE
110 WINDOW 0,2,0,2
120 REM  DRAW A BOX AROUND WINDOW
130 AXIS
140 AXIS 0,0,2,2
150 REM  PRINT A CHARACTER STRING
160 REM   POSITIONED AT LEFT EDGE OF WINDOW
170 MOVE 0,1
180 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
190 HOME
200 END

VIEWPORT 40,90,40,90
RUN
```

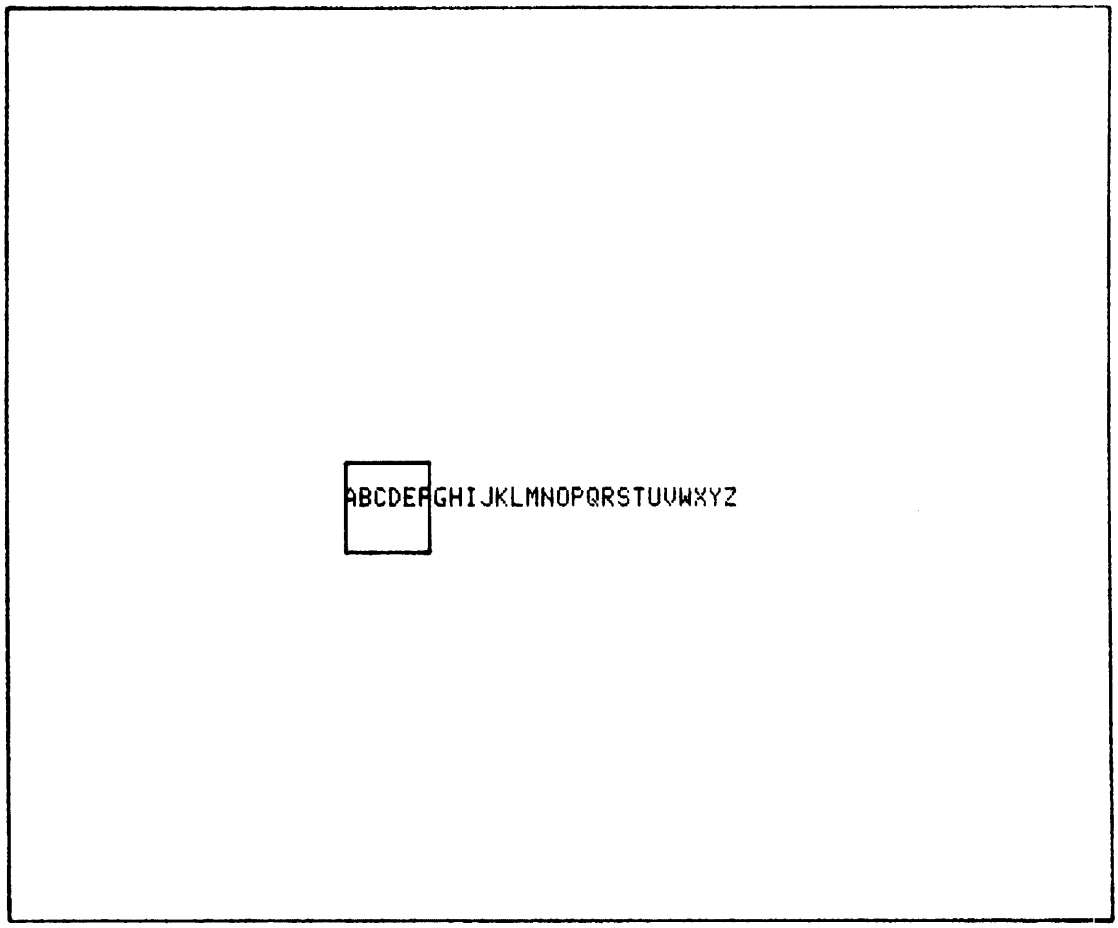
Here is the resulting output:



Now enter the following statements into the GS:

```
VIEWPORT 40,50,40,50
RUN
```

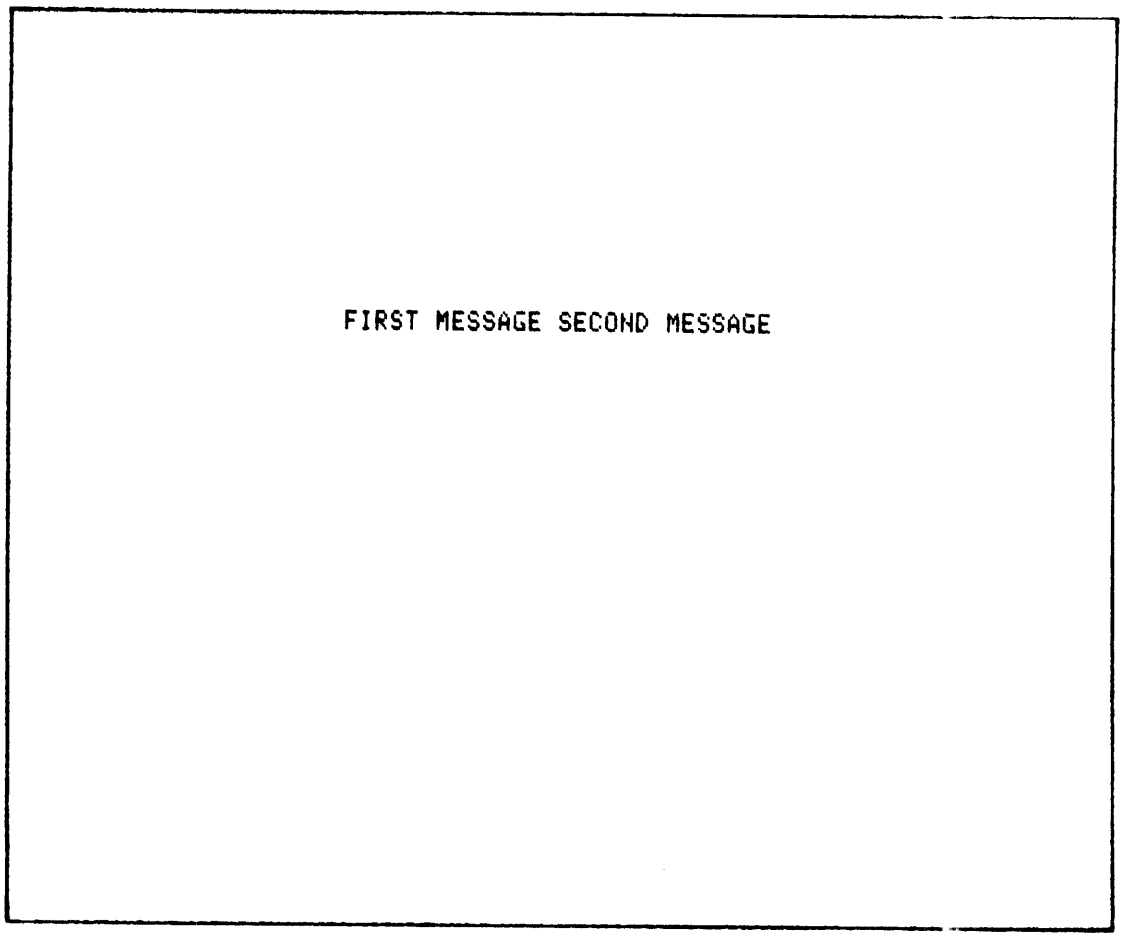
Here is the output that is produced:



The first viewport is 40 GDU's wide. It contains the printed characters easily. The second viewport is only 10 GDU's wide. Although the viewport has become smaller, the characters have remained the same size. They no longer fit into the smaller viewport. The window has remained unchanged. When the viewport shrinks, the amount of display space occupied by a user data unit also shrinks.

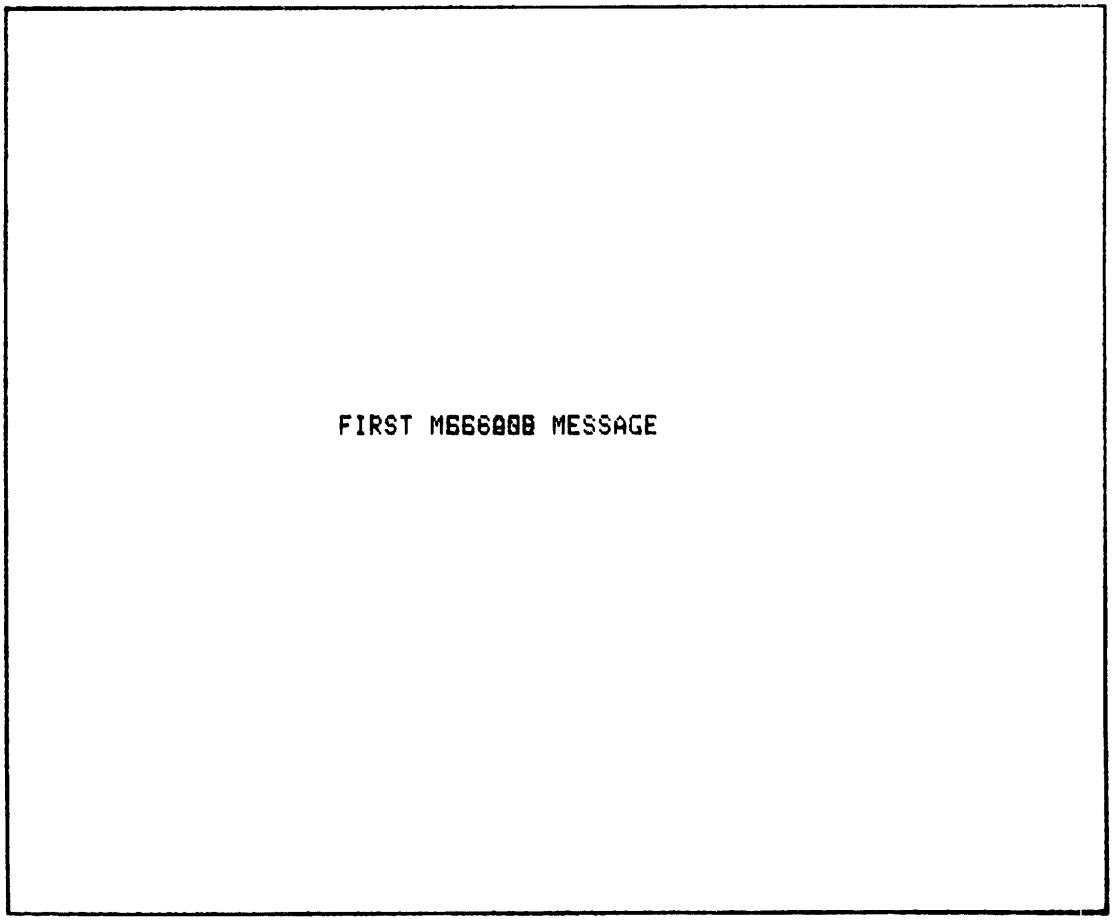
The following four examples illustrate the effect of the WINDOW and VIEWPORT commands on character output. The initial pair of examples demonstrate the effect of changing the window size while holding the viewport size constant. The first example is shown below:

```
100 PAGE
110 VIEWPORT 40,90,40,90
120 MOVE 0,1
130 PRINT "FIRST MESSAGE";
140 MOVE 1,1
150 PRINT "SECOND MESSAGE";
160 HOME
170 END

WINDOW 0,2,0,2
RUN
```



The two character strings above are separated by a comfortable distance. If the width of the window is doubled, that is if the viewport's horizontal dimension is made to contain twice as many user data units as above, notice what happens. (The result is shown below.)

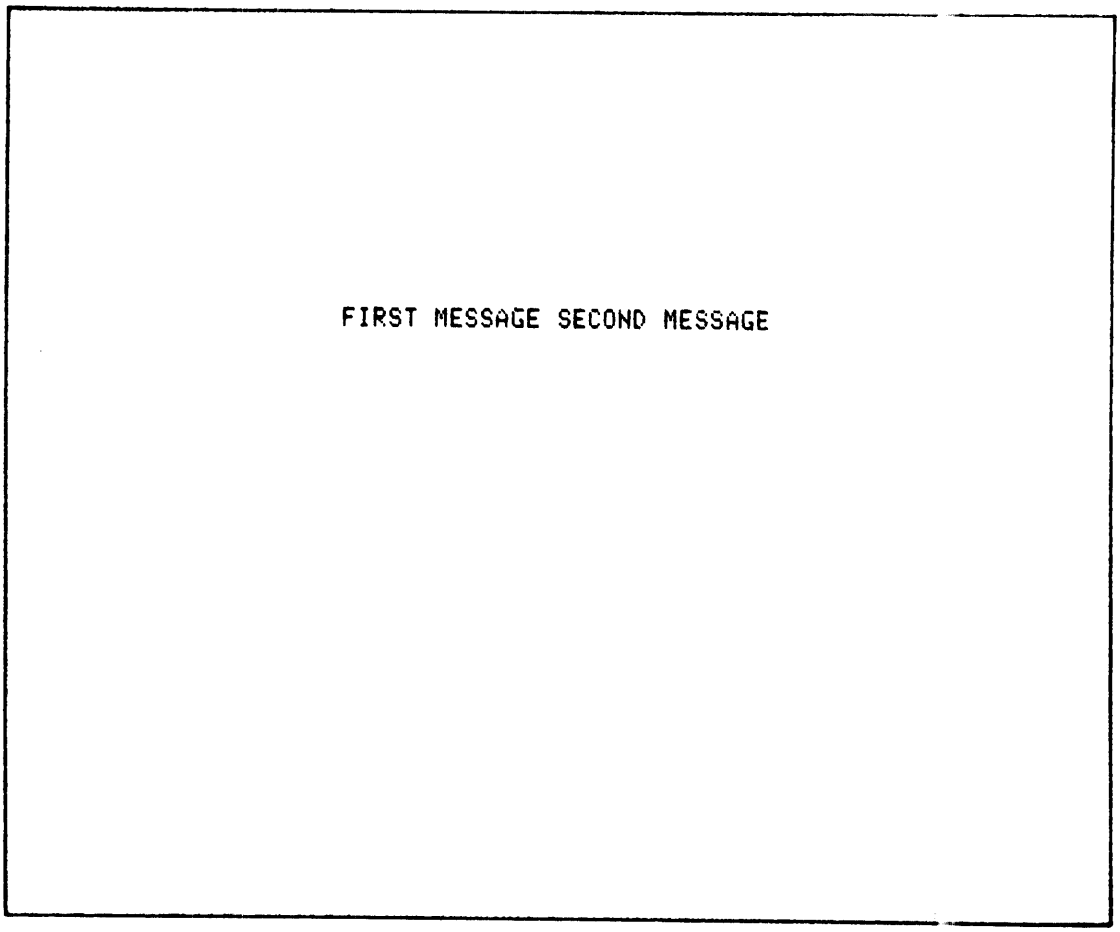
```
WINDOW 0,4,0,4
RUN
```



In the above example, the distance between the starting points of the two character strings (one user data unit in this case) now becomes less than the length of the first string. As a result, the two character strings now overlap. In any graphing application, this would be unacceptable.

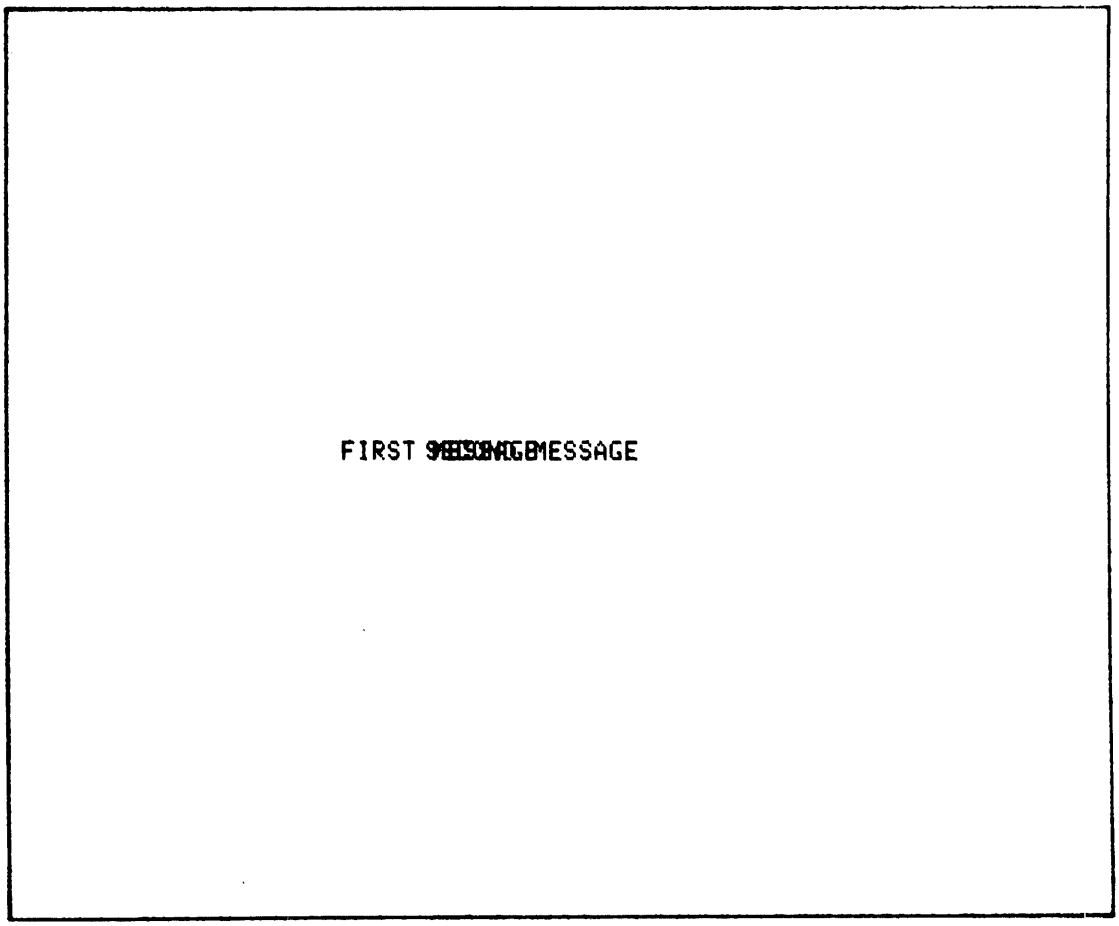
The next two examples show the effect of holding the window size constant and varying the viewport. The first example is essentially a return to the previous case, where the strings do not overlap. Enter the following statements:

```
110 WINDOW 0,2,0,2
VIEWPORT 40,90,40,90
RUN
```



The viewport is reduced to less than half its size in the above example by entering the following statements:

```
VIEWPORT 40,60,40,60
RUN
```



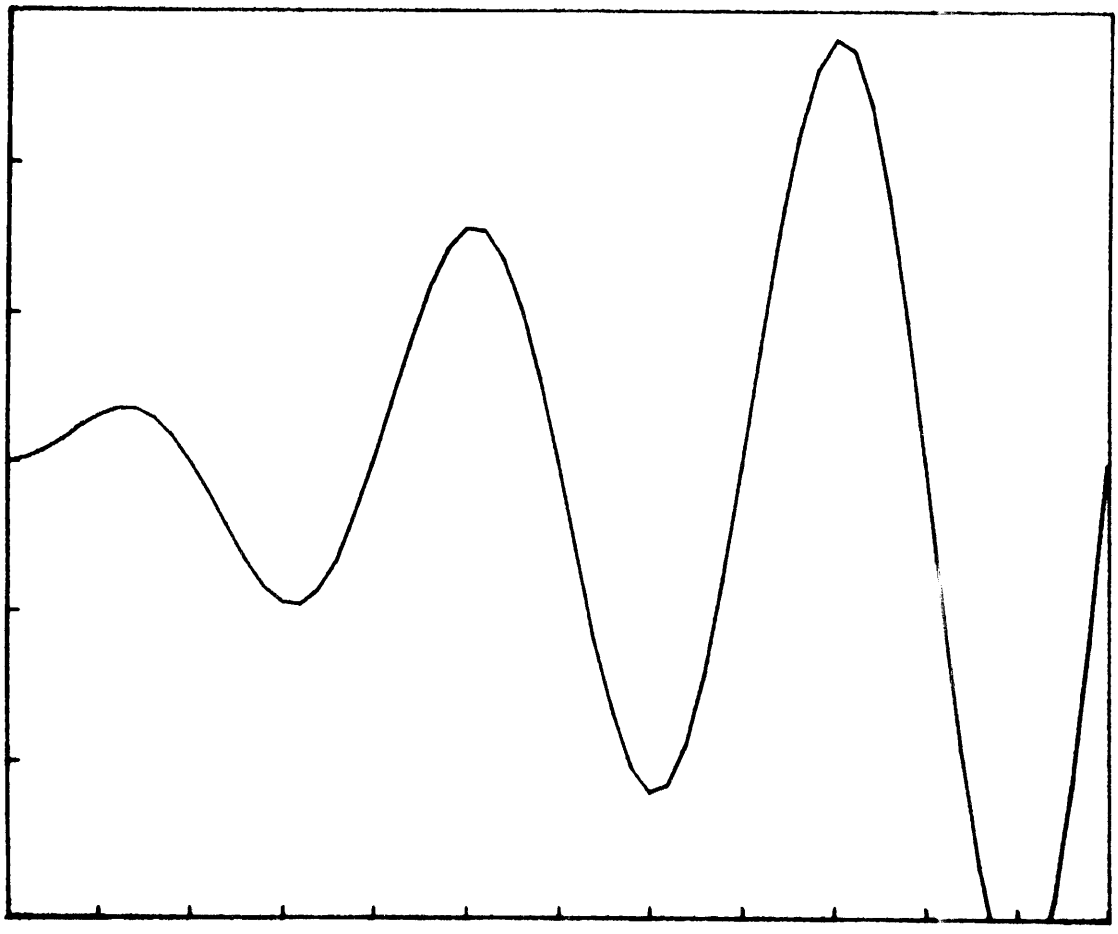
In the above example, the same number of user data units now occupy half the display space as before. The distance between the starting points of the two character strings, although being the same distance in user data space, is now half what it was before. So the second character string begins before the first one has ended. If the titling and labeling on a graph within a certain window and viewport is satisfactory, the same titling and labeling might be too sparse, too crowded, or totally unreadable if the window or viewport dimensions change.

TITLING

Below is a sample function graph with a basic set of axes.

The program will be gradually expanded as various kinds of titles and labels are added to it.

```
100 PAGE
110 INIT
120 DATA 0,130,0,100
130 READ V1,V2,V3,V4
140 VIEWPORT V1,V2,V3,V4
150 WINDOW 0,PI*6,0,30
160 AXIS PI/2,5
170 MOVE 0,15
180 FOR I=PI/10 TO PI*6 STEP PI/10
190 DRAW I,SIN(I)*I+15
200 NEXT I
210 HOME
220 END
```



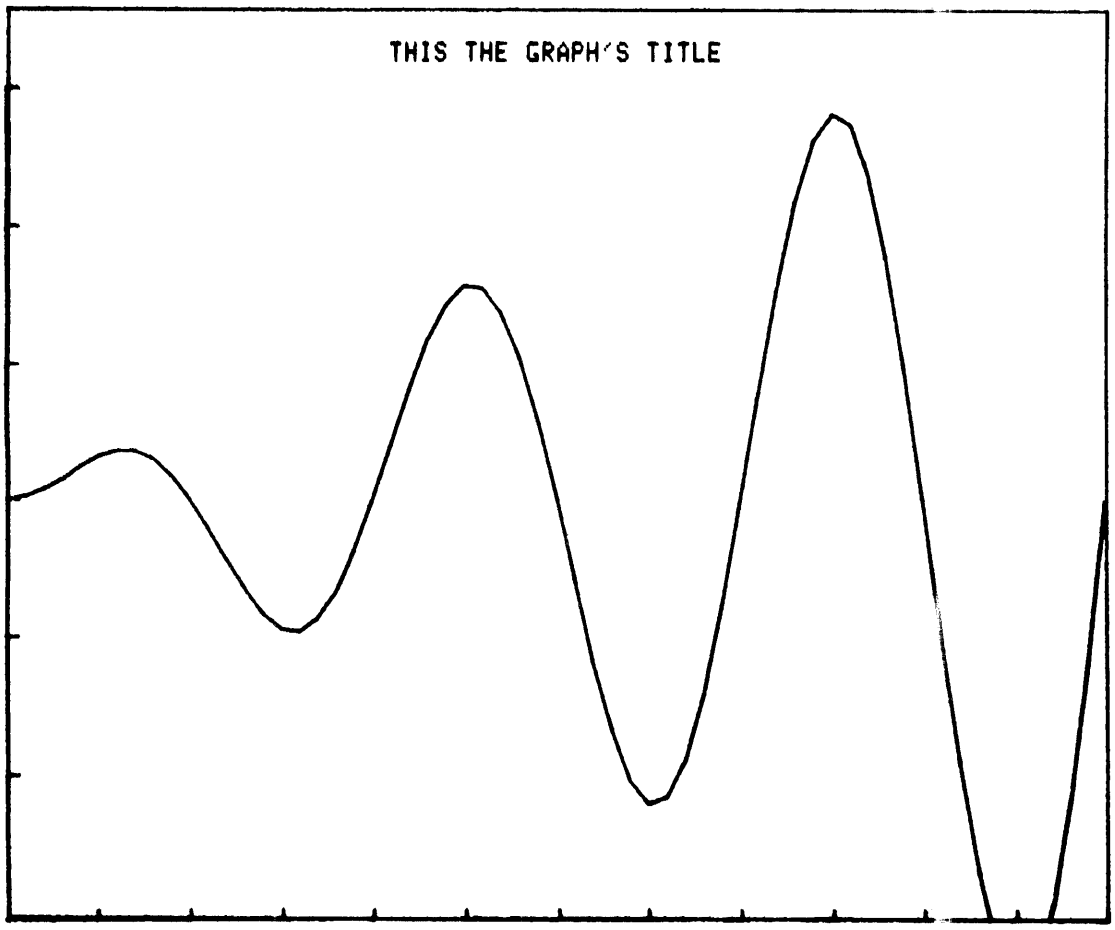
Adding a title above this graph (or any graph) is a three step process:

1. Reserve the space for the title
2. Properly position the beginning of the title
3. Print the title

Each of these steps can be performed in a variety of ways.

In the graph shown above, the data and axes totally fill the screen because of the VIEWPORT parameters. Therefore, no characters can be printed outside the specified viewport area. However, the top edge of the viewport can be lowered to allow room for the title. The statements shown below perform that function when added to the previous example program.

```
    .
    .
    .
140 VIEWPORT V1,V2,V3,V4-3*2.82
    .
    .
    .
210 REM  ADD TITLE TO GRAPH
220 A$="THIS THE GRAPH'S TITLE"
230 MOVE 3*PI,30
240 PRINT "K";
250 FOR I=1 TO LEN(A$)/2
260 PRINT "H";
270 NEXT I
280 PRINT A$;
290 HOME
300 END
    .
    .
    .
```



The MOVE command at statement 230 (above) places the graphic point in the middle of the horizontal data range and at the top of the vertical data range. The space between this point and the top of the allowable viewport (the vertical location specified by variable V4) is three character heights. This space was reserved by the VIEWPORT command in statement 140. Therefore, a move up of one character height will vertically center a line of characters within that reserved space. Statement 240 does just that: moves the graphic point up one character space. Centering the title horizontally is just a matter of moving the title's beginning point to the left by half the number of characters in the title, much as a title is centered on a typewritten page. The FOR . . . NEXT loop in statements 250 through 270 does this. It prints a number of backspace characters (CTRL H) equal to half the number of characters in the title string. The graphic point is now positioned at the correct location for the beginning of the title string.

How much room should be reserved for the title string? It depends mainly on how many lines of characters the title will have. The distance between adjacent lines of characters is approximately 2.82 GDU's. This is the smallest vertical space that should be reserved for one line of characters. In the above example, three lines of space are reserved, one for the title, one for a space above the title and one for a space below the title. This is 2.82*3 or about 8.5 GDU's. This change is implemented in the VIEWPORT command at line 140, above. This change lowers the height of the viewport which actually contains the data curve and axes, leaving 8.5 GDU's of vertical space between the top of this adjusted viewport and the top of the display. This unadjusted top is specified by the variable V4 and is the highest allowable location for anything to appear on the screen.

The title was positioned using a MOVE command and two control characters: CTRL K for moving up and CTRL H for moving left. Below is a summary of the characters which move the graphic point in increments of one character space.

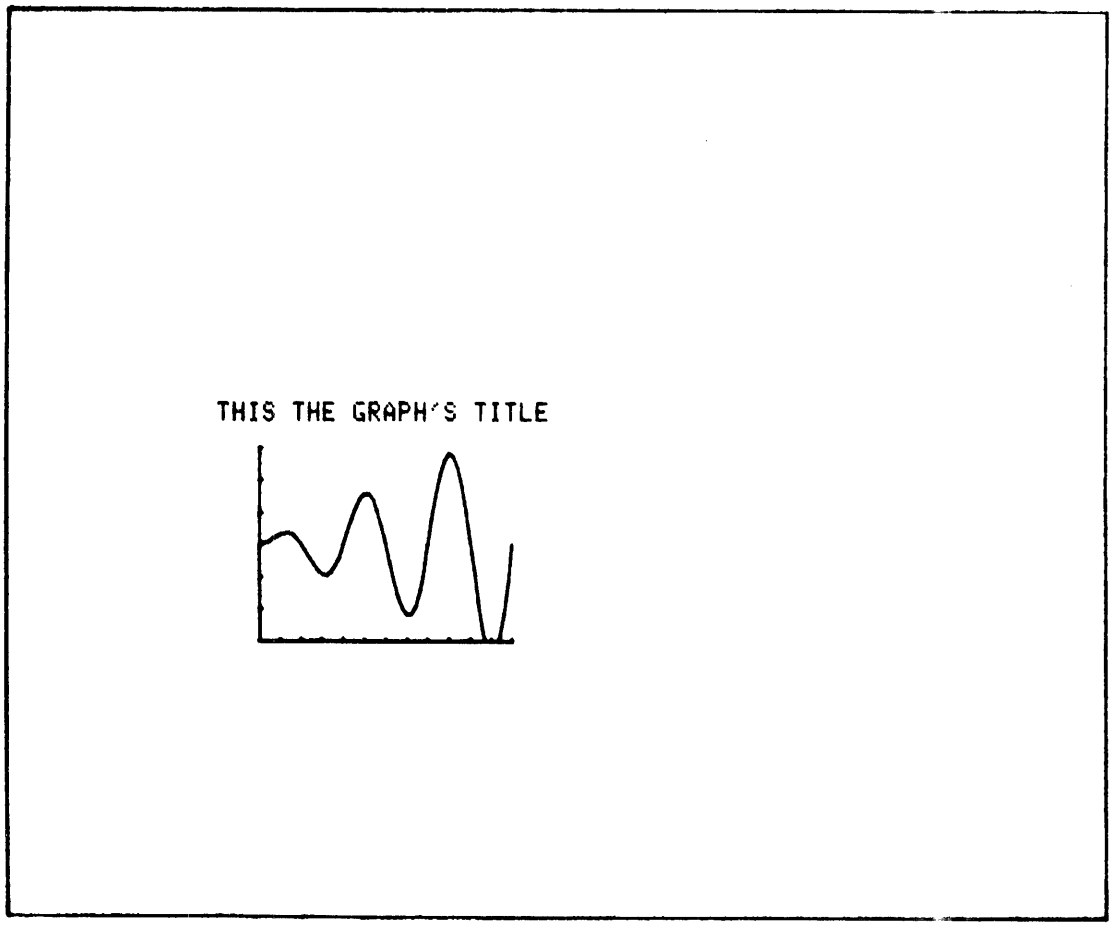
Action: Moves	Character Name	Obtainable From Key-board With	How to Place Into A Character String	ASCII Value
To the right	SPACE	SPACE BAR	A$ = " "	32
To the left	BACKSPACE	CTRL H	A$ = "H"	8
Down	LINE FEED	CTRL J	A$ = "J"	10
Up	VERTICAL TAB	CTRL K	A$ = "K"	11

Three of these characters are entered as control characters: Control H, Control J, and Control K.

This means that in order to enter the character into the GS, the CTRL key is used much like the SHIFT key. A control character is entered by pressing the desired character key while simultaneously pressing and holding the CTRL key. The "space" character (entered with the SPACE BAR) moves the graphic point to the right by 1.79 GDU's, the distance which separates adjacent characters. The "backspace" character (which prints on the display as "H" or "Control H") moves the graphic point to the left by the same distance. The "line-feed" character (which prints on the display as "J" meaning "Control J") moves the graphic point down by 2.82 GDU's, the distance which separates adjacent lines of characters. The "vertical tab" character (which prints on the display as "K" or "Control K") moves the graphic point up by the same distance. These distances are all GDU's and are not affected by the WINDOW, VIEWPORT, or SCALE statements. When manipulating the graphic point in this way, it must be remembered that the graphic point is moved to the right whenever a non-control character is output.

Centering the title within the viewport does not guarantee that the viewport is wide enough to contain the title. In the following example, the viewport is narrower than the title.

```
    .
    .
    .
120 DATA 30,60,30,60
    .
    .
    .
    .
    .
    .
```



It is easy to add a test which allows the title to be printed only when it will fit. Statement 225 performs such a test.

```
      .
      .
      .
220 A$="THIS THE GRAPH'S TITLE"
225 IF 1.79*LEN(A$)>V2-V1 THEN 290
      .
      .
      .
290 HOME
300 END
      .
      .
      .
```

The test at statement 225 compares the amount of horizontal space the title will occupy (1.79 GDU's times the number of characters in A$) with the width of the viewport (V2-V1). If the title is wider than the viewport, the title is not printed.

AXIS LABELS

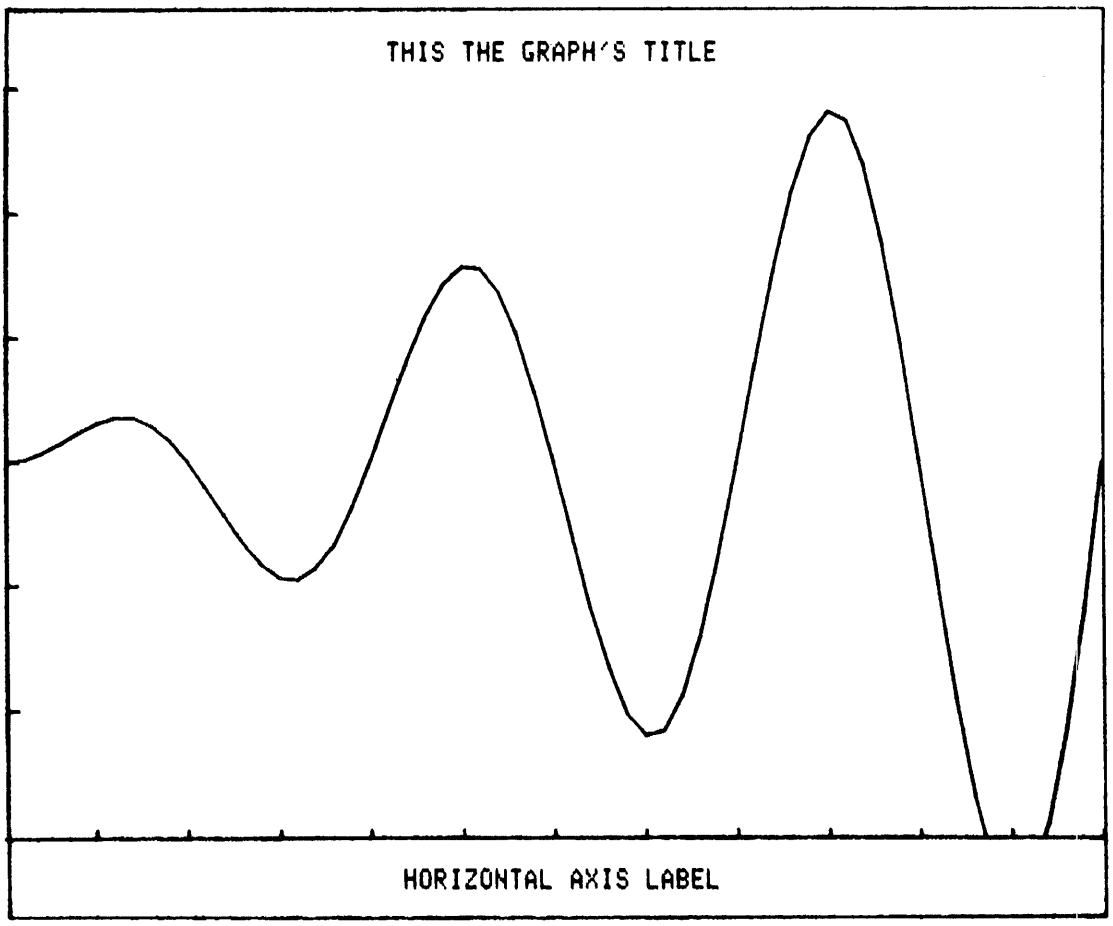
Horizontal Axis Label

The same three steps are used when labeling an axis:

1. Reserve space.
2. Position the beginning of the character string.
3. Print the string.

Shown below are the additional statements required to add a label to the horizontal axis.

```
      .
      .
      .
120 DATA 0,130,0,100
      .
      .
      .
140 VIEWPORT V1,V2,V3+3*2.82,V4-3*2.82
      .
      .
      .
290 REM  ADD HORIZONTAL AXIS LABEL
300 H$="HORIZONTAL AXIS LABEL"
310 MOVE 3*PI,0
320 PRINT "JJ";
330 FOR I=1 TO LEN(H$)/2
340 PRINT "B";
350 NEXT I
360 PRINT H$;
370 HOME
380 END
      .
      .
      .
```



The space is reserved in the same manner as before except that the bottom of the viewport is raised. This is done by forming an adjusted viewport with a bottom 3*2.82 GDU's above the previously specified value. In statement 140, 3*2.82 is added to V3, the third argument of the viewport statement. The additional statements (290 through 380) are almost identical to those added for the title. There is one exception: line 320. Execution of statement 310 places the graphic point at the middle of the bottom edge of the window. In order to center the character line in the provided space, the graphic point must be moved down two lines. This is because the graphic point is at the lower left corner of the dot matrix which forms the characters. So statement 320 (above) outputs two line feed characters. The other statements function exactly the same as the corresponding ones which place the graph title on the display.

Vertical Axis Label

When labeling the vertical axis, being able to print a character string vertically saves much space on the display. The following is an example of how this is done.

(This program can be entered and run without affecting the program being developed if the statement numbers are carefully chosen.) Enter the following statements:

```
800 PAGE
810 INIT
820 DIM A$(15),X$(1)
830 A$="THIS IS A TEST"
840 PRINT A$
850 GOSUB 1000
860 C=1930+50
870 PRINT C
880 A$=STR(C)
890 PRINT A$
900 GOSUB 1000
910 END
1000 FOR I=1 TO LEN(A$)
1010 X$=SEG(A$,I,1)
1020 PRINT X$;"␣";
1030 NEXT I
1040 RETURN
RUN 800
```

Here is the output produced:

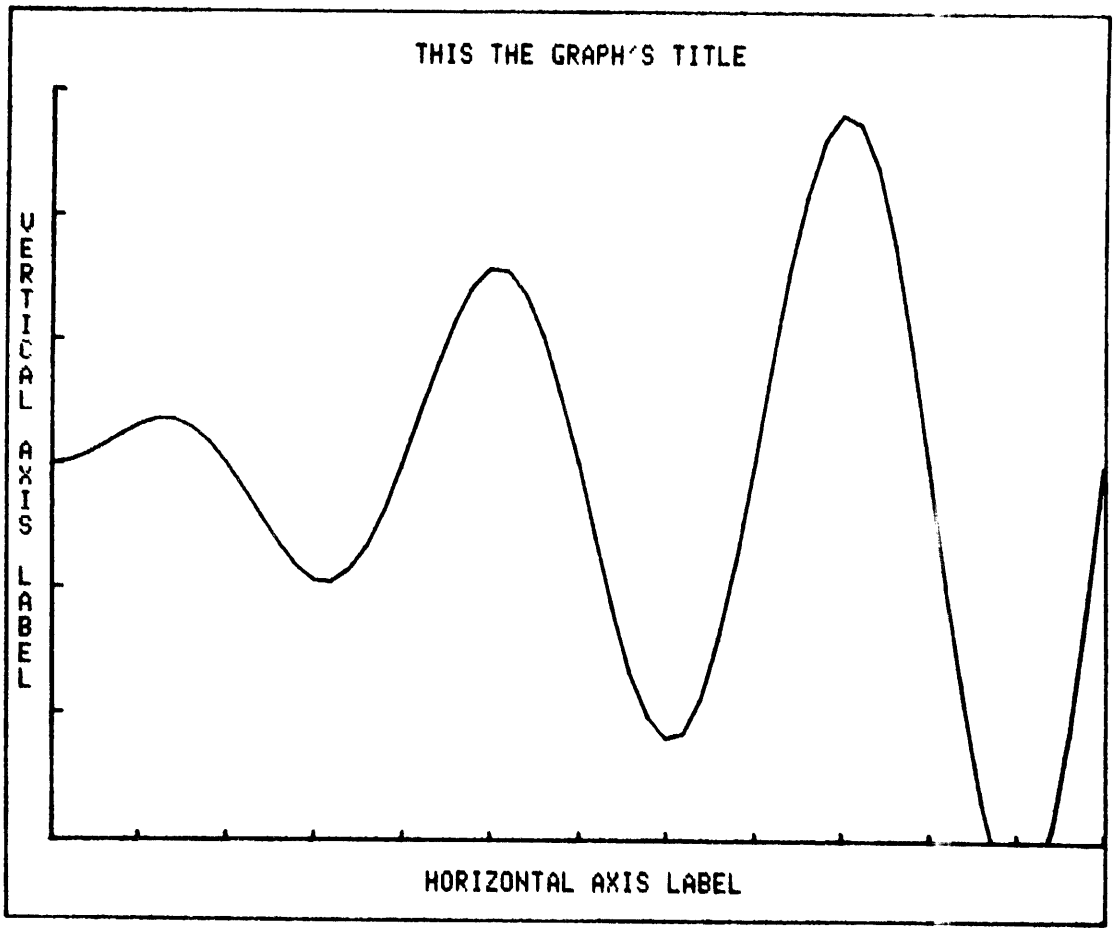
```
THIS IS A TEST
T
H
I
S

I
S

A

T
E
S
T
 1980
 1980

1
9
8
0
```

The DIM command at statement 820 is optional. While its use saves memory, its presence in the above example is mainly to show that X$ only has to be one character long. The SEG command (statement 1010) extracts a character at a time from A$. After each character is extracted, it is printed and followed by a backspace (CTRL H) and a line feed (CTRL J). This same process can be done easily to numbers by using the STR function.

The STR function (statement 880) converts the number in C into a character string. It can then be placed into A$ and printed vertically, just as the string "THIS IS A TEST" was printed vertically. The subroutine which begins at statement 1000 simply prints the string A$ but in a vertical orientation.

A vertical axis label can be added with the following statements:

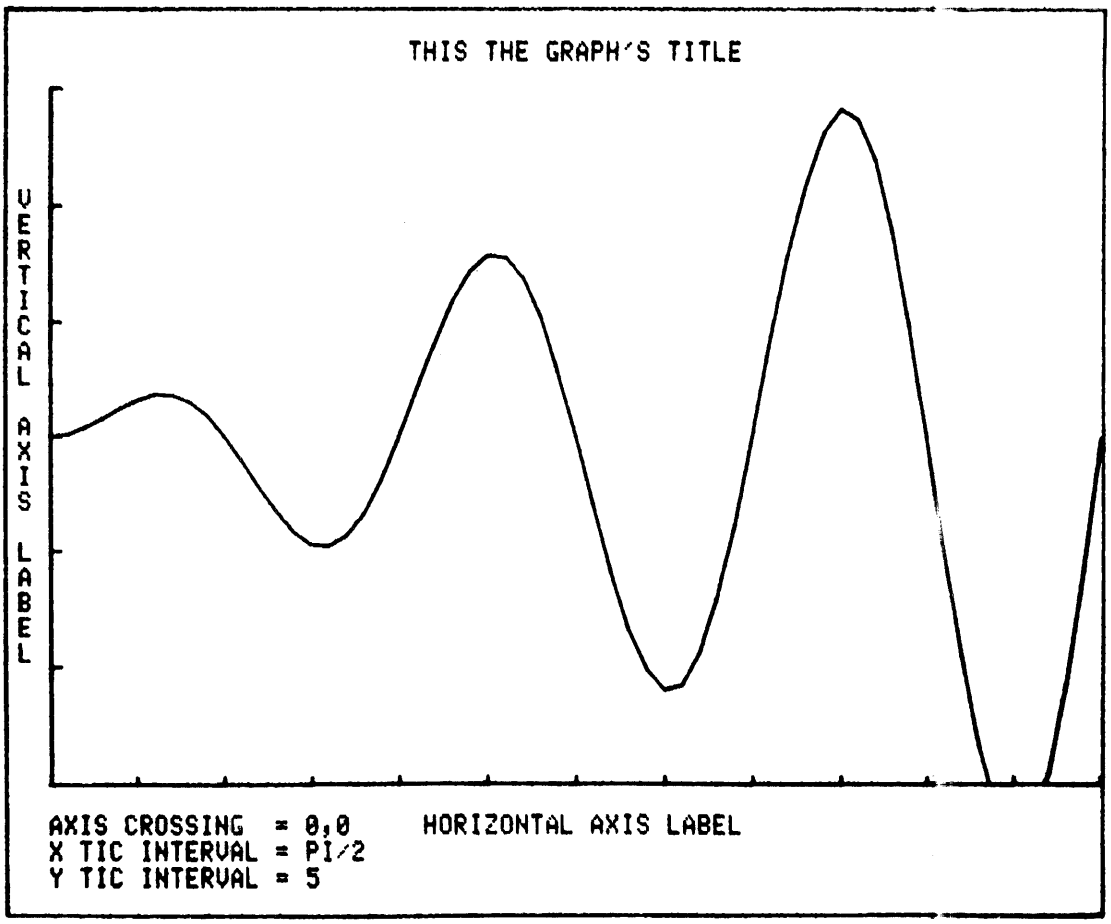
```
DELETE 800,1040
    .
    .
    .
140 VIEWPORT V1+3*1.79,V2,V3+3*2.82,V4-3*2.82
    .
    .
    .
370 REM  ADD VERTICAL AXIS LABEL
380 V$="VERTICAL AXIS LABEL"
390 MOVE 0,15
400 PRINT "ⒷⒷ";
410 FOR I=1 TO LEN(V$)/2
420 PRINT "Ⓚ";
430 NEXT I
440 FOR I=1 TO LEN(V$)
450 X$=SEG(V$,I,1)
460 PRINT X$;"ⒷⒿ";
470 NEXT I
480 HOME
490 END
    .
    .
    .
```


THIS THE GRAPH'S TITLE
VERTICAL AXIS LABEL
HORIZONTAL AXIS LABEL

TIC MARK LABELS

Labeling individual tic marks, though not difficult, requires more programming effort and GS memory space than is appropriate for many applications. It is sometimes preferable just to print the parameters of the AXIS command somewhere within the viewport. The most suitable location for this information is usually just above the lower edge of the viewport, either to the right or left of the horizontal axis label. If only one line is being printed, the space reserve specified at the third parameter of the VIEWPORT statement (line 140 above) need not be changed. However, in this next example, three lines are printed for the AXIS and tic information. So five lines of space are reserved.

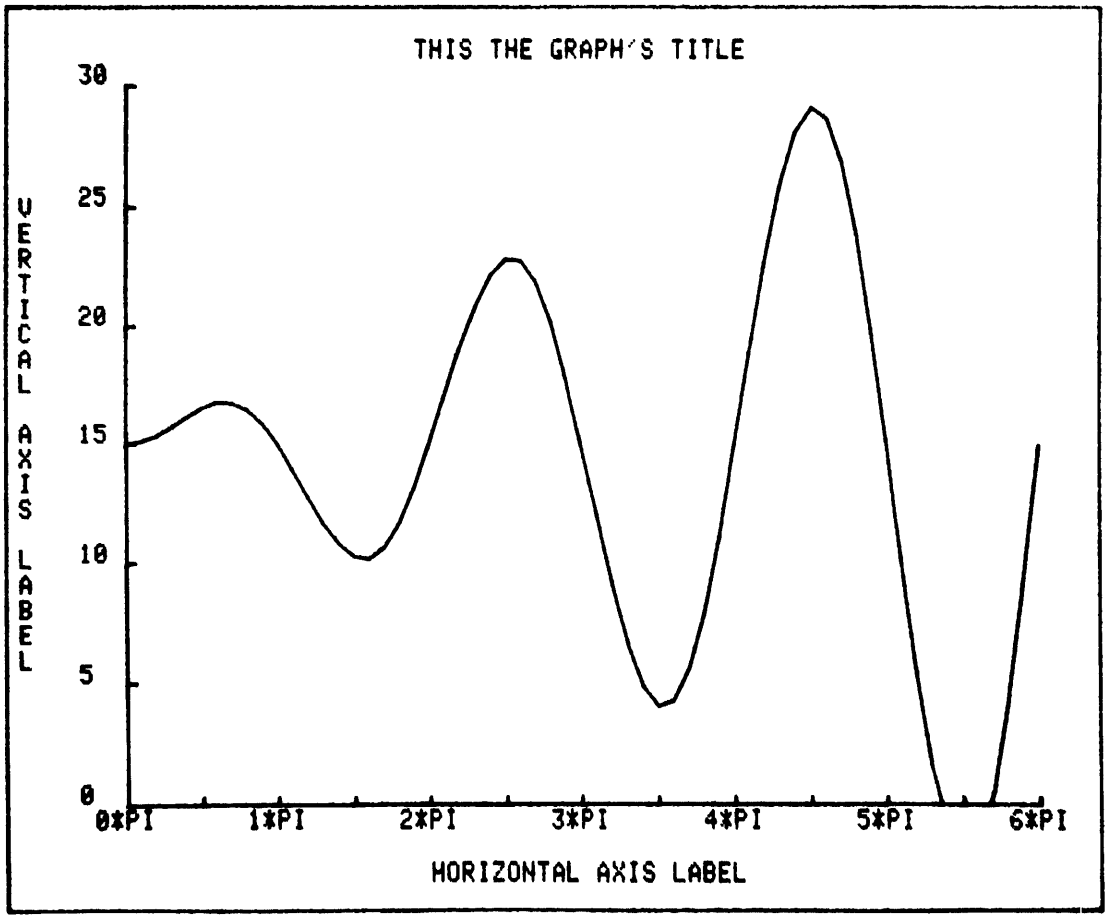
```
    .
    .
    .
140 VIEWPORT V1+3*1.79,V2,V3+5*2.82,V4-3*2.82
    .
    .
    .
480 REM  PRINT TIC & AXIS DATA
490 MOVE 0,0
500 PRINT "␣␣AXIS CROSSING  = 0,0";
510 MOVE 0,0
520 PRINT "␣␣␣X TIC INTERVAL = PI/2";
530 MOVE 0,0
540 PRINT "␣␣␣␣Y TIC INTERVAL = 5";
550 HOME
560 END
    .
    .
    .
```



A technique which makes the graphed data easier to comprehend is to place a value corresponding to each tic adjacent to the actual tic mark itself. Below is an example of this method, including a list of all statements that were changed or added to the example above.

```
    .
    .
    .
140 VIEWPORT V1+8*1.79,V2-4*1.79,V3+4*2.82,V4-3*2.82
    .
    .
    .
320 PRINT "JJJ";
    .
    .
    .
400 PRINT "BBBBBBB";
    .
    .
```

```
.
480 REM LABEL HORIZ. AXIS TIC MARKS
490 FOR I=0 TO 6*PI STEP PI
500 MOVE I,0
510 PRINT "ШШ┘";I/PI;"*PI";
520 NEXT I
530 REM  LABEL VERT. AXIS TIC MARKS
540 FOR I=0 TO 30 STEP 5
550 MOVE 0,I
560 PRINT "ШШ";I;
570 NEXT I
580 HOME
590 END
.
.
.
```



The VIEWPORT statement (line 140) is changed because more space is required for the labels. Eight character widths are reserved to the left (V1+8*1.79). This is larger than before because space must be reserved for more characters. Four character widths are reserved to

the right (V2–4*1.79) to make room for the last tic label on the end of the horizontal axis. Five character heights are reserved below the graph. More space is needed here mainly to make the labels easier to read. No additional room has been reserved above the graph in this example because nothing else has been added to that area.

Because of these changed space allocations, the initial positioning of both the horizontal axis label (statement 320) and the vertical axis label (statement 400) are also changed. If they are not changed, the axis labels will conflict with the tic labels printed in statements 480 through 590.

Statements 490 through 520 label the horizontal axis tics. Inherent in the VIEWPORT command at line 140 are assumptions about the amount of space occupied by these tic labels. One character height of space below the axis is reserved, along with four character widths to the right of the last tic mark on the axis. If this last space is not reserved, part of the last label is printed outside the viewport, an unacceptable situation. An assumption about the width of the labels is implied in two statements: in the STEP size of the FOR command at statement 490 (if this step size is too small, the labels will be printed on top of each other), and in the number of backspaces printed at statement 510. The number I/PI will always be an integer in this situation because of the FOR . . . NEXT loop's beginning value and step size.

The "*PI" in statement 510 (above) shows that the addition of characters to each tic label requires only the presence of the desired character string in the PRINT statement. The combination of a one digit integer and the "*PI" is always four characters long. Hence the two backspace characters (CTRL H) in statement 510.

Unless special considerations are made in the program, it must be assumed that each tic label is made up of no more than a certain number of characters. Only after this assumption is made can the step size in the FOR statement (statement 490 in this example) and the label position (statement 510 in this example) be set up. The step size in the FOR statement also determines the frequency of the tic labels. If the tic mark labels are too close together, they become more difficult to read.

Most of the horizontal axis tic label description applies with only slight changes to the vertical axis tic labels. The major difference is that the assumed maximum for the number of characters allowable in a vertical axis tic label has a more direct effect on the STEP size in the FOR statement (statement 540). The first parameter of the VIEWPORT at line 140 was changed to V1+8*1.79 (eight character widths reserved). The assumed maximum number of characters for the label is three. There are several ways to ensure that the number of characters in a given label will not exceed the assumed maximum. These are described later in this section.

RESERVING SPACE

As was stated earlier in this section, the three steps required for adding characters to a graph are:

1. Reserving space
2. Positioning
3. Printing

The first of these three, space reservation, is now discussed.

The technique used so far in all this section's examples is to adjust the parameters of the viewport command by the appropriate number of GDU's in each direction. This is the best general method because the amount of space reserved is always known exactly. Another approach is to alter the VIEWPORT by specified percentages of the VIEWPORT height and width. The example below decreases the size of the viewport so that the adjusted rectangle is smaller than the unadjusted one.

Unadjusted:

```
VIEWPORT V1,V2,V3,V4
```

Percentage Adjustment:

```
VIEWPORT V1+(V2-V1)*0.1,V2-(V2-V1)*0.1,V3+(V4-V3)*0.1,V4-(V4-V3)*0.1
```

Advantages: Space reservation changes with viewport size; a more pleasant appearance results.

Disadvantages: The exact quantity of space reserved is unknown.
Too little space is reserved with small viewports, too much with large ones.

Both of these techniques have the characteristic of adjusting the size of the clipping rectangle, that area outside which no graphic output will appear.

The size of the window can also be adjusted to make room for labels. This technique is not appropriate for the above example. However, it is described and used in the next section, where its use is very appropriate.

POSITIONING

The second required process is to position labels within the allotted space. After the graphic point is placed at the location to be labeled, it must then be moved a certain number of character spaces to ensure that the printed characters are positioned properly. The following situations are among those which require this capability: centering a graph title within the allotted space, properly positioning a tic mark label, and centering a character over a point on the display. There are three ways this can be done.

The simplest way is to use the control characters described in Sections 1 and 6: SPACE, BACKSPACE (CTRL H), LINE FEED (CTRL J), and VERTICAL TAB (CTRL K). All examples so far in this section have used this technique. The following program fragment shows its use:

```
    .
    .
    .
300 REM  TO MOVE ONE CHARACTER SPACE HEIGHT DOWN
310 PRINT "J";
320 REM  TO MOVE ONE CHARACTER SPACE HEIGHT UP
330 PRINT "K";
340 REM  TO MOVE ONE CHARACTER SPACE WIDTH TO THE RIGHT
350 PRINT " ";
360 REM  TO MOVE ONE CHARACTER SPACE WIDTH TO THE LEFT
370 PRINT "H";
    .
    .
    .
```

Advantages: Simple to use.
Doesn't require keeping track of WINDOW and VIEWPORT parameters.
It is a true relative move in character distance increments.

Disadvantages: Can only move in increments of one entire character height or width (is therefore too imprecise for some situations).

The second way is to totally rescale the graph at the point desired, use RMOVE to position the graphic point, and re-window to restore the original mapping situation. The example below moves 1.3 character widths to the left and .2 character heights below the point X,Y.

```
    .
    .
    .
300 WINDOW W1,W2,W3,W4
310 MOVE X,Y
320 SCALE 1,1
330 RMOVE -1.3*1.79,-0.2*2.82
340 PRINT "LABEL";
350 WINDOW W1,W2,W3,W4
    .
    .
    .
```

In the preceding program fragment, the SCALE command at statement 320 establishes a one-to-one correspondence between user data units and GDU's. It is also possible to establish a one-to-one correspondence between user data units and character spaces. The number of character widths and heights to be moved can then be used directly as arguments of the RMOVE command (statement 330). The program fragment below demonstrates this technique showing the appropriate arguments to use in the SCALE command.

```
      .
      .
      .
300 WINDOW W1,W2,W3,W4
310 MOVE X,Y
320 SCALE 1/1.79,1/2.82
330 RMOVE -1.3,-0.2
340 PRINT "LABEL";
350 WINDOW W1,W2,W3,W4
      .
      .
      .
```

Advantages: Allows movement in fractional character increments.
Simple to use.

Disadvantages: The previously defined window must be restored after this technique is used (as in statement 350, above).

The third way is to use the window and viewport parameters to compute the number of horizontal and vertical user data units required for a movement on the display of 1 GDU. Below is an example illustrating this technique.

```
      .
      .
      .
300 WINDOW W1,W2,W3,W4
310 VIEWPORT V1,V2,V3,V4
320 REM  S1 IS NUMBER OF HORIZONTAL USER DATA UNITS PER GDU
330 S1=(W2-W1)/(V2-V1)
340 REM  S2 IS NUMBER OF VERTICAL USER DATA UNITS PER GDU
350 S2=(W4-W3)/(V4-V3)
360 REM  TO THE LEFT 2.4 CHARACTER WIDTHS AND
370 REM          DOWN 1.5 CHARACTER HEIGHTS
380 RMOVE -2.4*S1*1.79,-1.5*S2*2.82
      .
      .
      .
```

Advantages: Allows movement of fractional character increments.

Disadvantages: Can be cumbersone if window or viewport specifications are intricate.
Requires keeping track of WINDOW and VIEWPORT parameters.
Can require two extra variables.

PRINTING

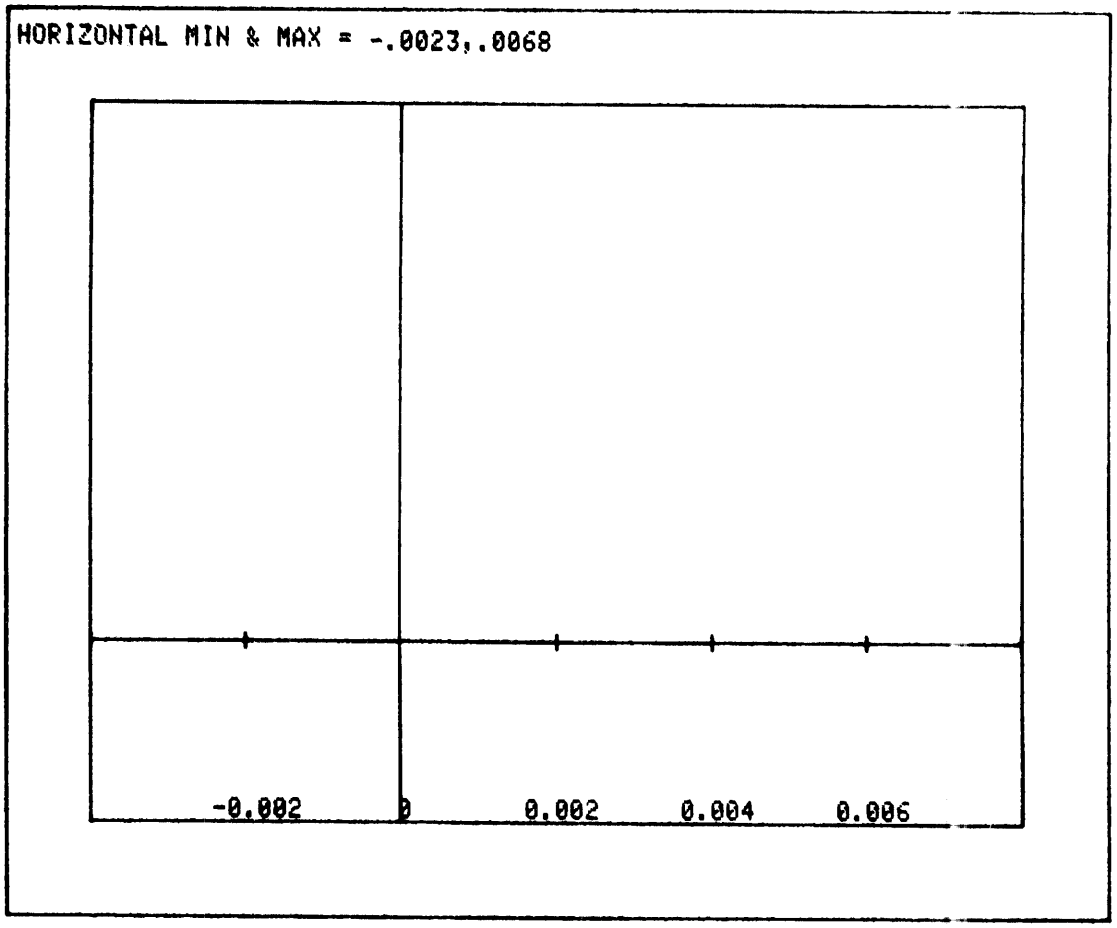
The third required process is to actually print the label. The most useful enhancement for printing is the guarantee that a number's printed image on the screen will not exceed a certain number of characters. There are several ways to provide this guarantee. One way is to use the INTeger function to round the number to a specified quantity of significant digits. Another method is to utilize PRINT . . . USING, documented in the Plot 50 Introduction to Programming in BASIC. These techniques can ensure that numbers of excessive length are never printed. An example which utilizes the PRINT . . . USING command is described in the next section (Section 7).

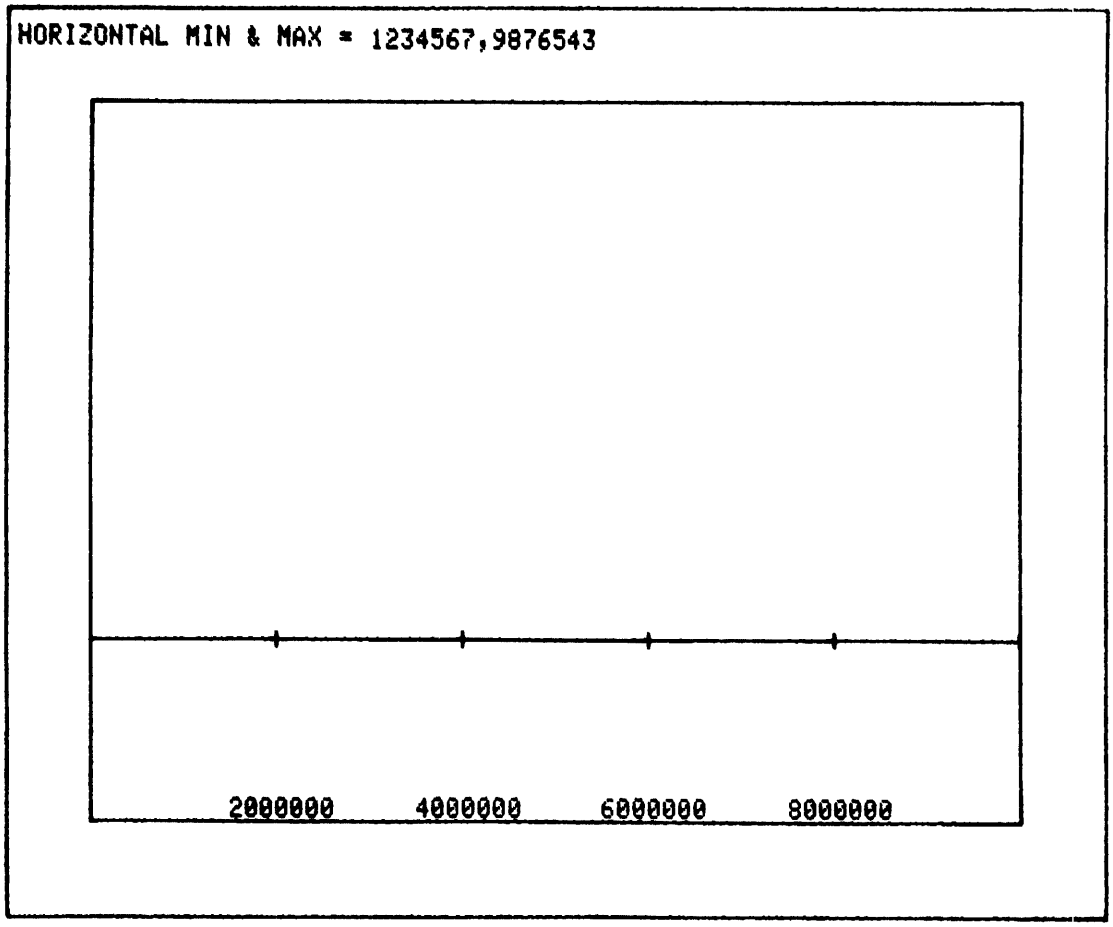
Section 7

ENHANCEMENTS

"NEAT" TIC INTERVALS

Manually calculating the appropriate tic intervals for a given data range can be tedious. A preferable situation is to have a program do it. Below are two example graphs with "neat" tic intervals on the horizontal X axis calculated by the Graphic System.





The program used is listed below.

```
100 REM  SAMPLE LABELED "NEAT" TICS FOR HORIZONTAL AXIS
110 PAGE
120 DATA 10,120,10,90,-130,130,-25,75,6
130 RESTORE
140 READ V1,V2,V3,V4,W1,W2,W3,W4,N
150 VIEWPORT V1,V2,V3,V4
160 HOME
170 PRINT "HORIZONTAL MIN & MAX = ";
180 INPUT W1,W2
190 REM   CALCULATE "NEAT" INTERVAL & EXPANDED DATA LIMITS
200 GOSUB 2000
210 WINDOW M1,M2,W3,W4
220 AXIS 0,0,M1,W3
230 AXIS 0,0,M2,W4
240 AXIS S,0
250 GOSUB 3000
260 END
```

```
2000 REM  COMPUTE "NEAT" TIC INTERVAL FOR HORIZONTAL AXIS
2010 REM  R="RAW" TIC INTERVAL
2020 R=(W2-W1)/N
2030 REM S=LARGEST INTEGER POWER OF TEN STILL SMALLER THAN RAW INTERVAL
2040 S=10↑INT(LGT(R))
2050 REM T= RAW INTERVAL / NEXT SMALLEST POWER OF TEN
2060 T=R/S
2070 IF T>2 THEN 2110
2080 IF T=1 THEN 2150
2090 S=2*S
2100 GO TO 2150
2110 IF T>5 THEN 2140
2120 S=5*S
2130 GO TO 2150
2140 S=10*S
2150 REM  S="NEAT" TIC INTERVAL
2160 REM  ADJUST DATA MINIMUM
2170 M1=INT(W1/S)
2180 M1=S*(M1+2)
2190 IF M1<W1 THEN 2220
2200 M1=M1-S
2210 GO TO 2190
2220 REM  FOUND ADJUSTED MINIMUM
2230 REM  ADJUST DATA MAXIMUM
2240 M2=INT(W2/S)
2250 M2=S*(M2-2)
2260 IF W2<M2 THEN 2290
2270 M2=M2+S
2280 GO TO 2260
2290 REM  FOUND ADJUSTED MAXIMUM
2300 RETURN

2990 REM  LABELING ROUTINE FOR HORIZONTAL AXIS
3000 S7=M1
3010 S8=W3
3020 S1=M2-S/2
3030 S7=S7+S
3040 IF S7>S1 THEN 3120
3050 MOVE S7,S8
3060 A$=STR(S7)
3070 FOR Q=1 TO LEN(A$)/2
3080 PRINT "B";
3090 NEXT Q
3100 PRINT S7
3110 GO TO 3030
3120 RETURN
```

A subroutine beginning at line 2000 calculates an appropriate tic interval for the horizontal axis based on the three parameters: the horizontal data minimum (assigned to variable W1), the horizontal data maximum (assigned to W2), and the maximum number of tic marks desired (assigned to N). The tic interval calculated is a multiple of 1, 2, or 5. Tic intervals calculated in this manner are referred to as "neat". N is initialized to 6 in the READ statement at line 140. Lines 2020 through 2150 calculate a tic interval which will result in no

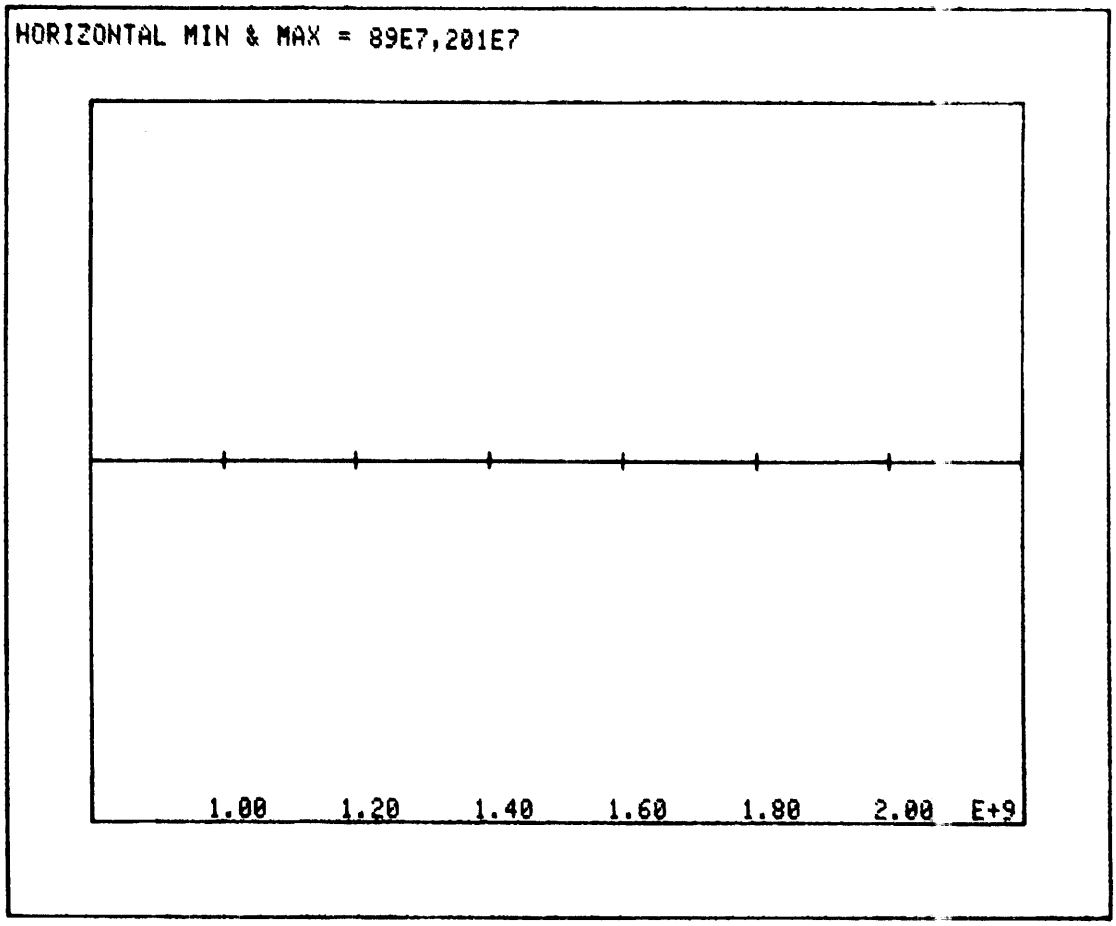
more than N tic intervals being drawn. This interval is passed back to the main program in variable S. Lines 2160 through 2220 adjust the data minimum downward so that the new data minimum will be an integer multiple of the tic interval. This adjusted data minimum is passed back in variable M1 leaving the original data minimum, W1, undisturbed. Similarly, lines 2230 through 2290 adjust the data maximum upward, passing the new data maximum back in variable M2.

After the tic interval, adjusted data maximum and adjusted data minimum have been determined, the window can be defined with the WINDOW command (line 210). Note that the arguments for the horizontal range of the WINDOW command are the adjusted data minimum (M1) and the maximum (M2), not the original data minimum (W1) and maximum (W2). The two AXIS statements at lines 220 and 230 simply draw a box around the window to show that no tic labels extend outside it. The AXIS statement at line 240 draws the axis with the "neat" tic interval S. Then the subroutine beginning at line 3000 is called to label the horizontal axis.

This routine to label the tics requires the following information: the adjusted data minimum (M1) to know where to start; the adjusted data maximum (M2) to know where to stop; the tic interval (S); and the vertical data minimum (W3) to know where to position the labels vertically. All other variables used are scratch variables.

The subroutine positions the labels with the same technique that was used at the end of the previous section. Line 3050 places the graphic point directly below the tic to be labeled. The value to be printed is converted to a character string using the STR function. Then a FOR. . . NEXT loop (statements 3070 through 3090) outputs a number of backspaces equivalent to half the number of characters in the label to be printed. If a situation arises where the labels are too close together and begin to overlap, changing N to a smaller number will cause fewer labels to be printed, reducing the chance of overlap. In the above program, space for the labels is reserved by expanding the window. The entire program requires no access to the viewport parameters. If the viewport is specified from the keyboard just prior to program execution, there is no need for any VIEWPORT command (such as the one at line 140) in the program at all.

For certain applications, tic labels with a normalized scientific notation are desirable. The example below shows such labels.

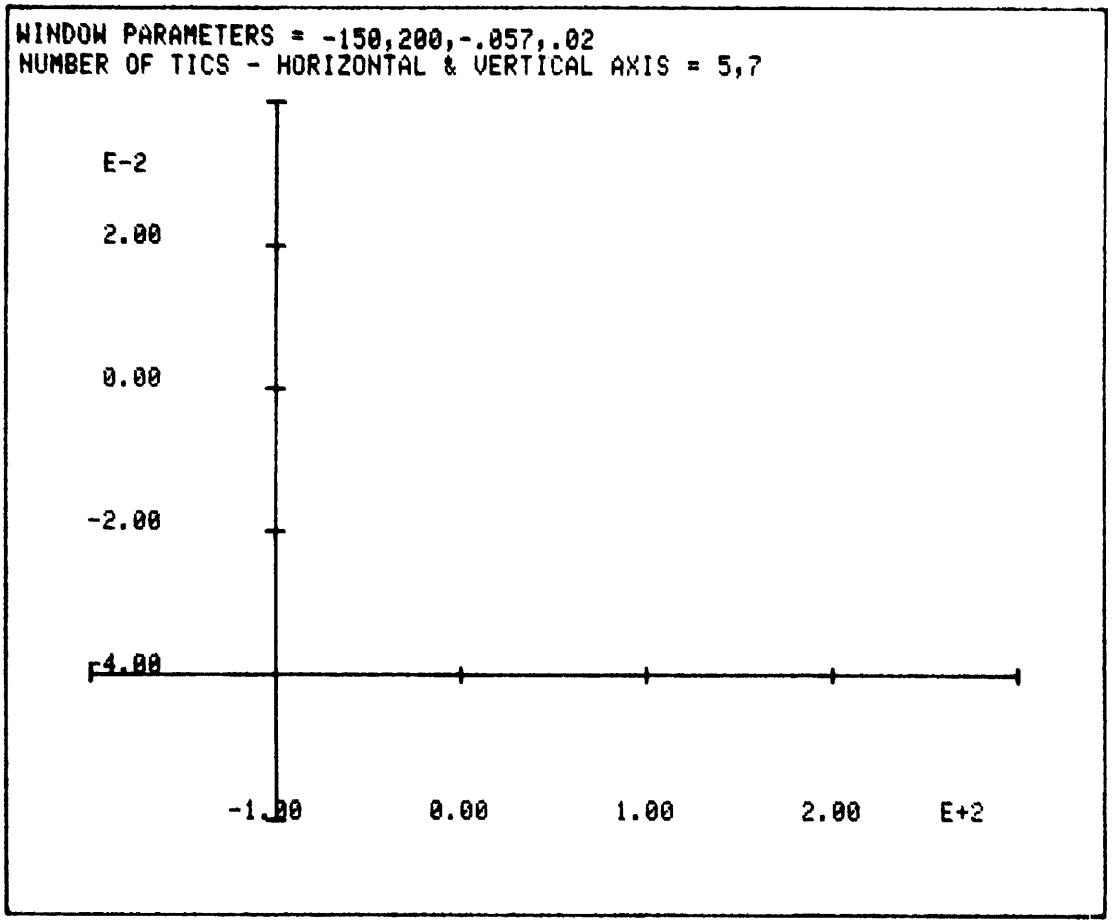


The program which generated these labels is identical with the previous example program, listed above, with the exception of the labeling subroutine beginning at statement 3000. It is listed below.

```
2990 REM  LABELING ROUTINE FOR HORIZONTAL AXIS
3000 S7=M1
3010 S8=W3
3020 S3=ABS(M1+S) MAX ABS(M2-S)
3030 S3=INT(LGT(S3)+1.0E-8)
3040 S2=10↑-S3
3050 S1=M2-S/2
3060 S7=S7+S
3070 IF S7>S1 THEN 3120
3080 MOVE S7,S8
3090 PRINT "ⅡⅡ";
3100 PRINT USING "-D.2D,S":S7*S2
3110 GO TO 3060
3120 IF S3=0 THEN 3160
3130 S7=S1
3140 MOVE S7,S8
3150 PRINT USING "2A,+FD,S":" E";S3
3160 RETURN
```

The required input variables are the same (M1, M2, W3, S). But the way they are handled is slightly different. Statements 3020 through 3040 calculate a normalizing factor, used when the tic values are printed (statement 3100) and to determine if an exponent needs to be printed (statement 3120 through 3150). The example program above prints the tic labels with a PRINT USING command. This guarantees that the tic labels never exceed a certain number of characters in length (4 + 1 space in this example). Therefore, the labels can be positioned properly with a constant number of backspaces. Statement 3090, which outputs two backspaces, replaces a FOR . . . NEXT loop. In addition, the task of reserving space for the labels is simplified by knowing for certain how many characters comprise each label.

The next example extends the techniques of the above example to the general case of labeling tics on both the horizontal and vertical axes.



```
100 REM  LABELED "NEAT" TICS - GENERAL CASE
110 PAGE
120 PRINT "WINDOW PARAMETERS = ";
130 INPUT W1,W2,W3,W4
140 PRINT "NUMBER OF TICS - HORIZONTAL & VERTICAL AXIS = ";
150 INPUT N1,N2
160 VIEWPORT 10,120,10,90
170 DIM P(8)
180 P(1)=W1
190 P(2)=W2
200 P(3)=N1
210 P(5)=W3
220 P(6)=W4
230 P(7)=N2
240 REM CALCULATE NEW LIMITS AND INTERVALS
250 P5=3
260 GOSUB 2000
270 P5=7
280 GOSUB 2000
290 WINDOW P(1),P(2),P(5),P(6)
300 AXIS P(3),P(7),P(1)+P(3),P(5)+P(7)
310 REM  LABEL THEM
320 P5=4
330 A$="###"
340 GOSUB 3000
350 P5=8
360 A$=""
370 GOSUB 3000
380 HOME
390 END


2000 REM  P(P5) = MINIMUM NO. OF TICS
2010 P1=(P(P5-1)-P(P5-2))/P(P5)
2020 P2=10↑INT(LGT(P1))
2030 P1=P1/P2
2040 IF P1>2 THEN 2080
2050 IF P1=1 THEN 2120
2060 P2=2*P2
2070 GO TO 2120
2080 IF P1>5 THEN 2110
2090 P2=5*P2
2100 GO TO 2120
2110 P2=10*P2
2120 REM  ADJUST DATA MIN
2130 P1=INT(P(P5-2)/P2)
2140 P3=P2*(P1+2)
2150 IF P3<P(P5-2) THEN 2180
2160 P3=P3-P2
2170 GO TO 2150
2180 P(P5-2)=P3
2190 REM  ADJUST DATA MAX
2200 P1=INT(P(P5-1)/P2)
2210 P3=P2*(P1-2)
2220 IF P(P5-1)<P3 THEN 2250
2230 P3=P3+P2
2240 GO TO 2220
2250 P(P5-1)=P3
2260 REM  P(P5)=ADJUSTED TIC INTERVAL
2270 P(P5)=P2
2280 RETURN
```

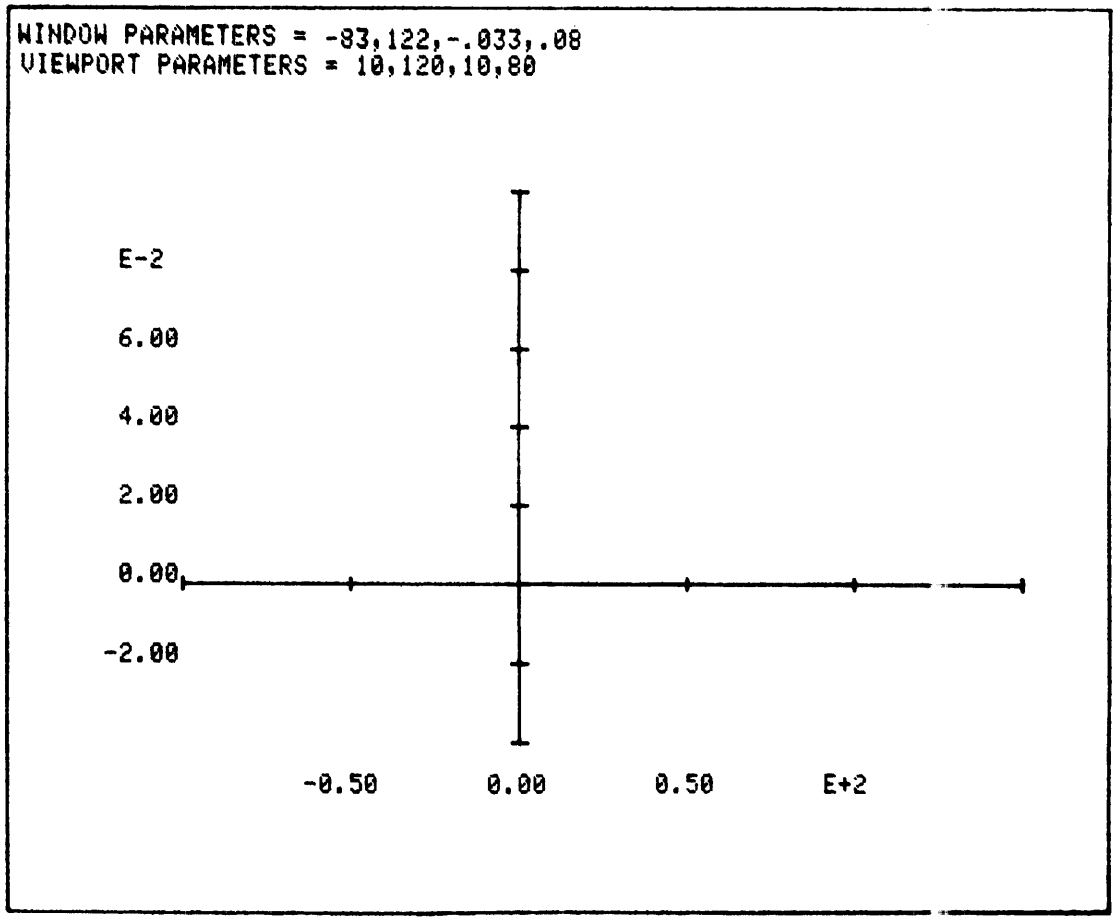
```
3000 REM  LABEL AXIS
3010 P4=P(P5-1)
3020 P(4)=P(1)
3030 P(8)=P(5)
3040 P3=ABS(P(P5-3)+P4) MAX ABS(P(P5-2)-P4)
3050 P3=INT(LGT(P3)+1.0E-8)
3060 P2=10↑-P3
3070 P1=P(P5-2)-P4/2
3080 P(P5)=P(P5)+P4
3090 IF P(P5)>P1 THEN 3140
3100 MOVE P(4),P(8)
3110 PRINT A$;
3120 PRINT USING "-D.2D,S":P(P5)*P2
3130 GO TO 3080
3140 IF P3=0 THEN 3180
3150 P(P5)=P1
3160 MOVE P(4),P(8)
3170 PRINT USING "2A,+FD,S":" E";P3
3180 RETURN
```

Neither of the routines in the program listed above requires any knowledge of what the viewport has been set to. The VIEWPORT statement at line 160 could be deleted and the viewport set from the keyboard just prior to program execution. (An INIT command anywhere in the program would restore the viewport parameters to the default, full size, values.) Lines 120 through 150 cause the data limits and the desired number of tics to be input from the keyboard. Statements 170 through 230 set up array P, which is used to pass values to and from the routines. (The values assigned to variables W1, W2, W3, W4, N1, and N2 remain unchanged after they are input.) Array P is set up to allow a subroutine to be usable for both the horizontal and vertical axes. The subroutines listed in the example above perform functions identical to those in the first two examples of this section. They have been changed only to allow them to be used on both the horizontal and vertical axes.

The routine beginning at line 2000 affects only the variables P(P5), P(P5-1), and P(P5-2). All other variables in the routine are used for scratch purposes. (The values they contain are of use only while the subroutine at statement 2000 is being executed.) When the routine at statement 2000 is called with P5 equal to 3 (as it is at statement 260), the routine affects only the elements P(1), P(2), and P(3) in array P. When the routine is called with P5 equal to 7 (as it is at statement 280), the routine affects only P(5), P(6), and P(7). When entering the routine with P5 equal to 3, meaning that all data values pertain to the horizontal axis, the subroutine uses P(1) as the original data minimum, P(2) as the original data maximum and P(3) as the specified number of tic marks. The routine uses these same locations to return the values it calculates. It returns the adjusted data minimum in P(1), the adjusted data maximum in P(2), and the "neat" tic interval in P(3). Similarly, when called with P5 = 7, the routine returns the adjusted data minimum P(5), the adjusted data maximum in P(6), and the "neat" interval in P(7). These values are then used directly in the WINDOW command in line 290 and the AXIS command in statement 300.

The routine beginning at line 3000 has data passed to it in a similar manner. There are, however, two significant changes. The labeling routine at 3000 prints A$ (statement 3110) before printing each label (statement 3120). A$ contains positioning characters as needed: three backspaces for horizontal axis labels, no characters at all for vertical axis labels. The other difference is that P(4) and P(8) are used to position each label, with P(4) being the horizontal position and P(8) being the vertical position. The variable P5 determines which of them is updated in the loop formed by statements 3080 through 3130. This scheme, which may appear complex, allows very efficient use of memory space for both data and program statements.

The next example (shown below) is identical to the above example with the following exceptions: lines 140 through 160, line 200, and line 230.



```
100 REM  LABELED "NEAT" TICS - GENERAL CASE
110 PAGE
120 PRINT "WINDOW PARAMETERS = ";
130 INPUT W1,W2,W3,W4
140 PRINT "VIEWPORT PARAMETERS = ";
150 INPUT V1,V2,V3,V4
160 VIEWPORT V1+6*1.8,V2,V3+3*2.8,V4
170 DIM P(8)
180 P(1)=W1
190 P(2)=W2
200 P(3)=INT((V2-(V1+6*1.8))/(8*1.8)) MAX 1
210 P(5)=W3
220 P(6)=W4
230 P(7)=INT((V4-(V3+3*2.8))/(3*2.8)) MAX 1
240 REM CALCULATE NEW LIMITS AND INTERVALS
250 P5=3
260 GOSUB 2000
270 P5=7
280 GOSUB 2000
290 WINDOW P(1),P(2),P(5),P(6)
300 REM  AXI  P(3),P(7),P(1)+P(3),P(5)+P(7)
310 AXIS P(3),P(7)
320 REM  LABEL THEM
330 P5=4
340 A$="ΩΩΩJJ"
350 GOSUB 3000
360 P5=8
370 A$="ΩΩΩΩΩ"
380 GOSUB 3000
390 HOME
400 END
```

These changes allow the program to determine the number of tic labels which will fit into the specified viewport. The number of desired tics on each axis does not have to be entered. After the viewport parameters are entered in line 150, the viewport can be set up. As in examples in Section 6, the effective viewport size is reduced to make room for the labels. The viewport's horizontal minimum is increased by 6 character widths.

```
160 VIEWPORT V1+6*1.8,V2,V3+3*2.8,V4
```

This adjustment is based on the assumption that the vertical axis labels will be no wider than 5 characters. The viewport's vertical minimum is increased by 3 character heights.

```
160 VIEWPORT V1+6*1.8,V2,V3+3*2.8,V4
```

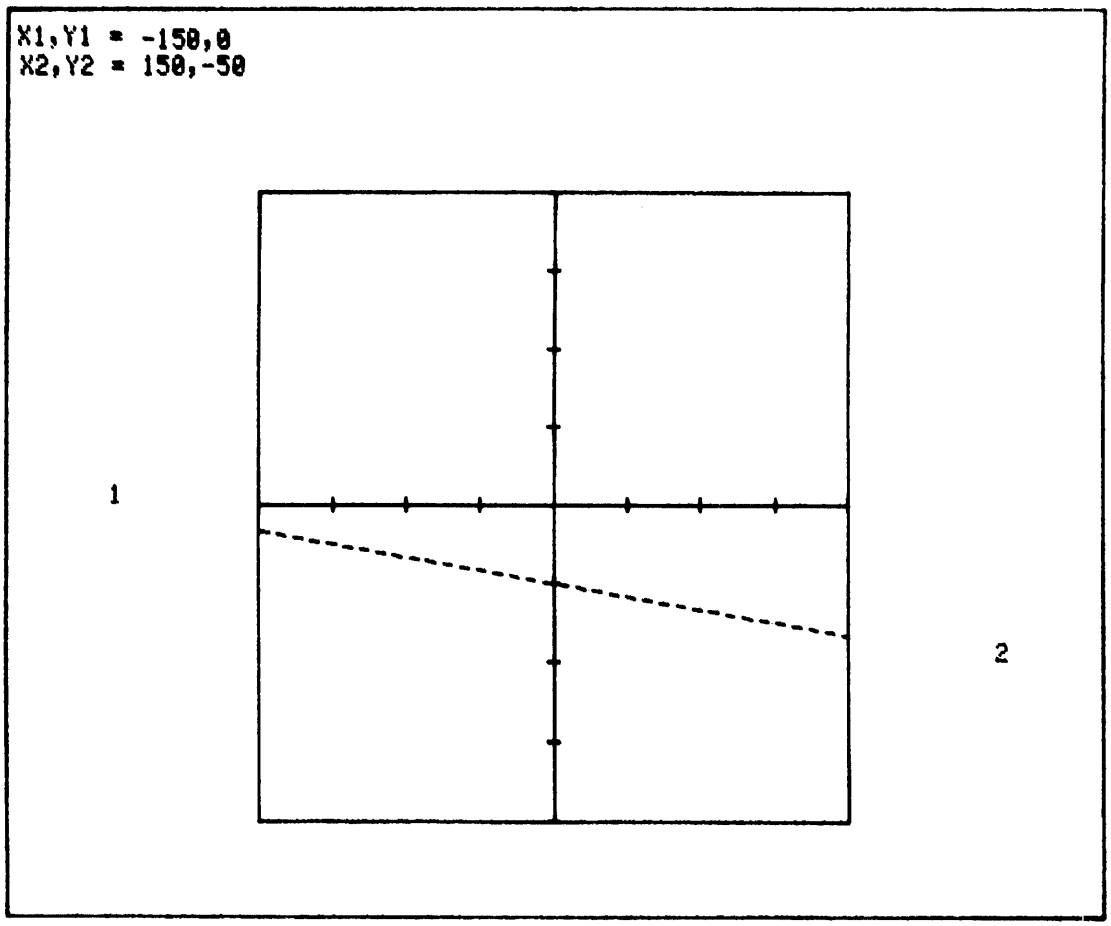
This adjustment is based on the assumption that the horizontal axis labels will occupy no more than one character line.

Line 200 determines the number of horizontal axis labels which will comfortably fit within the specified viewport width. It does this by dividing the effective viewport width [(V2 – (V1+6*1.8))] by the horizontal space taken up by 8 characters [8*1.8] . The figure 8 characters is used to ensure that the labels will not be too close together, causing a cluttered appearance. The MAX 1 in lines 200 and 230 prevents the number of labels requested from being less than 1, an event which the algorithm in the routine at 2000 is not prepared to handle.

Line 230 performs a similar task for the vertical axis labels. However, in this line, the effective viewport height is divided by 3 character heights to determine the appropriate number of labels. If the complete program is run with a variety of viewport sizes specified, the smaller viewports will produce fewer labels as appropriate.

DASHED LINES

Below is a sample graph of dashed lines.



The point X1,Y1 is located by the "1" to the left of the window. The point X2,Y2 is located by the "2" to the right of the window. The routine at statement 1000 draws a dashed line from X1,Y1 to X2,Y2, subject to the appropriate clipping performed by the window and viewport.

```
100 REM  DASHED LINE PROGRAM - GENERAL CASE
110 PAGE
120 RESTORE
130 DATA 30,100,10,80,-100,100,-100,100
140 READ V1,V2,V3,V4,W1,W2,W3,W4
150 VIEWPORT V1,V2,V3,V4
160 WINDOW W1,W2,W3,W4
170 REM DRAW A DASHED LINE FROM X1,Y1 TO X2,Y2
180 PRINT "X1,Y1 = ";
190 INPUT X1,Y1
200 PRINT "X2,Y2 = ";
210 INPUT X2,Y2
220 AXIS 25,25
230 REM  DRAW BOX AROUND VIEWPORT
240 AXIS 0,0,W1,W3
250 AXIS 0,0,W2,W4
260 MOVE X2,Y2
270 PRINT "2H";
280 MOVE X1,Y1
290 PRINT "1H";
300 GOSUB 1000
310 END

1000 REM  S1 IS NUMBER OF HORIZONTAL USER DATA UNITS PER GDU
1010 S1=(W2-W1)/(V2-V1)
1020 REM  S2 IS NUMBER OF VERTICAL USER DATA UNITS PER GDU
1030 S2=(W4-W3)/(V4-V3)
1040 REM   FIND HORIZONTAL DISTANCE IN GDU's
1050 D1=(X2-X1)/S1
1060 REM  FIND VERTICAL DISTANCE IN GDU's
1070 D2=(Y2-Y1)/S2
1080 REM  D IS DISTANCE BETWEEN POINTS IN GDU's
1090 D=SQR(D1↑2+D2↑2)
1100 IF D=0 THEN 1340
1110 REM  DESIRED DASH LENGTH IN GDU's IS L
1120 L=1
1130 REM  FIND HORIZONTAL COMPONENT OF DASH IN USER UNITS
1140 U1=L*(D1/D)*S1
1150 REM  FIND VERTICAL COMPONENT OF DASH IN USER UNITS
1160 U2=L*(D2/D)*S2
1170 MOVE X1,Y1
1180 REM CURRENT HORIZONTAL POINT IS H
1190 H=X1
1200 REM CURRENT VERTICAL POINT IS V
1210 V=Y1
1220 REM  CLEAR FLAG
1230 F=0
1240 REM  BEGIN LOOP TO DRAW DASHED LINE
1250 GOSUB 1350
1260 IF F=1 THEN 1330
1270 DRAW H,V
1280 GOSUB 1350
1290 IF F=1 THEN 1330
```

```
1300 MOVE H,V
1310 GO TO 1250
1320 REM  DONE
1330 DRAW X2,Y2
1340 RETURN
1350 REM  SUBROUTINE TO CHECK IF DONE
1360 H=H+U1
1370 V=V+U2
1380 IF ABS(H-X1)>ABS(X2-X1) THEN 1410
1390 IF ABS(V-Y1)>ABS(Y2-Y1) THEN 1410
1400 RETURN
1410 F=1
1420 RETURN
```

In order to draw the dashed line, the routine requires the following information: the viewport parameters (in this example V1, V2, V3, V4), the window parameters (in this example W1, W2, W3, W4), the desired dash length in GDU's (L in this example), and, of course, the beginning point (X1,Y2) and ending point (X2,Y2) of the dashed line. The subroutine is logically divided into two sections. The first section (from statements 1000 through 1160 in the above listing) uses the dash length, window and viewport information to calculate the horizontal and vertical components of the dash in user units. Once these two components are known, the dashed line can be drawn in user space, which makes the line subject to clipping.

This is very desirable, as the sample dashed line graph (above) shows. The second logical portion of the routine, beginning at statement 1170, is a loop which actually draws the dashed line. Since both the move and draw segments of the line need to be checked to see if the ending point of the line has been reached, this checking function has been placed in a subroutine beginning at line 1350. The variable F is a flag to tell the main loop in the routine if the ending point has been reached.

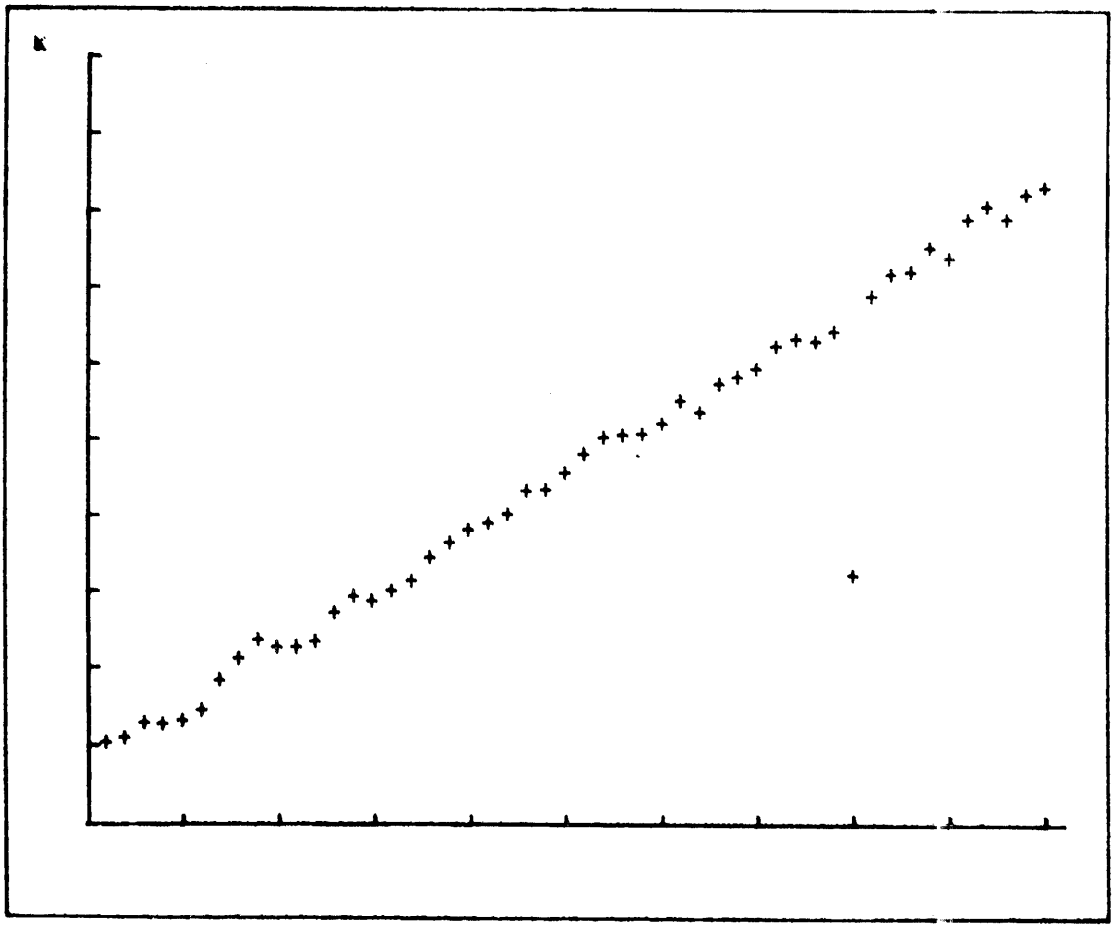
GRAPHIC DATA EDITING

The Graphic System can be used to edit data graphically. The program listed below shows a method for performing this task.

```
3 GO TO 100
4 RMOVE 0,P2
5 RETURN
8 RMOVE -P1,0
9 RETURN
12 RMOVE P1,0
13 RETURN
16 GO TO 260
20 REM  BEGIN EDITING
21 GOSUB 1000
22 PRINT @32,24:"|";
23 GO TO 22
24 RMOVE 0,-P2
25 RETURN
36 REM  FIND POINT
37 GOSUB 2000
38 RETURN
40 REM  CHANGE POINT
41 GOSUB 3000
42 RETURN
44 RMOVE 0,10*P2
45 RETURN
48 RMOVE -10*P1,0
49 RETURN
52 RMOVE 10*P1,0
53 RETURN
64 RMOVE 0,-10*P2
65 RETURN
97 REM    USER DEFINABLE KEYS:
98 REM      UP    LEFT    RIGHT    REGRAPH      EDIT
99 REM    DOWN                      FIND       CHANGE

100 INIT
110 SET KEY
120 REM  GENERATE SAMPLE DATA
130 DIM A(50)
140 M1=1.0E+300
150 M2=-1.0E+300
160 A(1)=10
170 FOR I=2 TO 50
180 A(I)=5*RND(-1)+5+I*1.5
190 IF I<>40 THEN 210
200 A(40)=A(39)/2
210 M1=M1 MIN A(I)
220 M2=M2 MAX A(I)
230 NEXT I
240 REM  GRAPH DATA
250 PAGE
260 VIEWPORT 10,125,10,95
270 WINDOW 0,51,0,100
280 AXIS 5,10
290 C1=51/120*1.55/2
300 C2=100/90*1.88/2
310 FOR I=1 TO 50
320 MOVE I-C1,A(I)-C2
330 PRINT "+H";
340 NEXT I
350 HOME
360 END
```

```
1000 REM  BEGIN EDITING
1010 REM  CALCULATE .5% OF WINDOW IN EACH DIRECTION
1020 P1=50*0.005
1030 P2=100*0.005
1040 REM   INITIAL PLACEMENT OF POINTER
1050 GIN G1,G2
1060 MOVE G1,G2
1070 REM  CHANGE FONT
1080 PRINT @32,18:5
1090 RETURN

2000 REM   FIND POINT
2010 GIN G1,G2
2020 D=1.0E+300
2030 REM  SEARCH DATA POINTS FOR CLOSEST FIT
2040 FOR I=1 TO 50
2050 D1=SQR((G1-I)↑2+(G2-A(I))↑2)
2060 IF D1>D THEN 2080
2070 D=D1
2080 NEXT I
2090 REM  TEST TO INSURE POINT REALLY FOUND
2100 J=INT(G1+0.5)
2110 IF J<1 OR J>50 THEN 2150
2120 IF ABS(J-G1)>2*P1 OR ABS(G2-A(J))>2*P2 THEN 2150
2130 REM  POINT FOUND
2140 RETURN
2150 REM  POINT NOT FOUND, RESTORE DEFAULT FONT
2160 PRINT @32,18:0
2170 END

3000 REM  CHANGE POINT
3010 REM   RESTORE FONT
3020 PRINT @32,18:0
3030 HOME
3040 PRINT "CHANGE A(";J;")"
3050 PRINT "FROM ";A(J)
3060 PRINT " TO  ";
3070 INPUT A(J)
3080 END
```

Typing RUN causes lines 100 through 360 to be executed. Lines 140 through 230 fill array A with sample data. The minimum and maximum values are placed in M1 and M2, respectively. Lines 190 and 200 cause an obviously out of line data point to be placed in A(40). Lines 250 through 340 graph the data in array A, marking the location of each data point with a "+". In order to be centered over the data point, each "+" is offset by a vertical distance equal to one half a character height and a horizontal distance equal to one half a character width. Line 290 calculates the distance, in horizontal user units, equal to one half character width. It divided the horizontal data range (51 in this example) by the width of the viewport in GDU's and multiplies the quotient by the character width in GDU's divided by two (1.55/2). The result, placed in C1, is half a character width in user units. The vertical offset distance is determined in a similar manner.



When the graph is drawn (shown above), it is seen immediately that a point is out of apparently normal range. In some applications, it is useful to select this data value for editing by merely pointing at it. Pressing user definable key 5 begins an editing mode which allows this to be done.

Whenever the editing mode is entered by pressing key 5, the arrow pointer appears on the display at the last position of the graphic point. After the program graphs the data in array A, the cursor is placed in the home position. If key 5 is pressed then, the pointer will appear in the home position in the upper left corner of the display. User keys 1, 2, 3, and 6 change the pointer's position. Key 1 moves it up; key 6 moves it down; key 2 moves it to the left; and key 3 moves it to the right. When each key is pressed in lower case, the pointer is moved .5% of the window size in the appropriate direction. When the key is pressed in upper case, the pointer is moved 5% of the window size in each direction. Holding any key down enters commands into the GS faster than they can be processed. It is best to move the arrow by tapping the desired key repeatedly, rather than by holding the key down. When the pointer

is as close as possible to the data point to be changed, press user key 9. This directs the program to search all the data values to determine if the pointer is acceptably close to a data value in the array. The pointer disappears while this search is executed. If the program fails to find a data value within 1% of the data ranges to the pointer, the program stops execution and the flashing rectangular cursor appears at the arrow's last location. If an acceptable match is found, the arrow reappears. Key 10 is now pressed to change the selected array element. In the upper left corner of the display is printed the index of the element selected, the element's current contents, and a request to input a new value. Pressing user key 4 regraphs the data, showing the position of the changed data point.

In several places, the program uses a non-default display secondary address. The secondary address 18, as used in lines 1080, 2160, and 3030, tells the display that one of the 6 character fonts in the GS display will be invoked.

The font determines what characters are actually printed on the display when certain ASCII character codes are sent to it. The normal font, called font number zero, is invoked with the following command: PRINT @32,18:0. There are six fonts, numbered 0 through 5. The arrow pointer, ASCII character code 124, is printed via font 5. Font 5 is invoked by line 1080 when the data edit mode is entered.

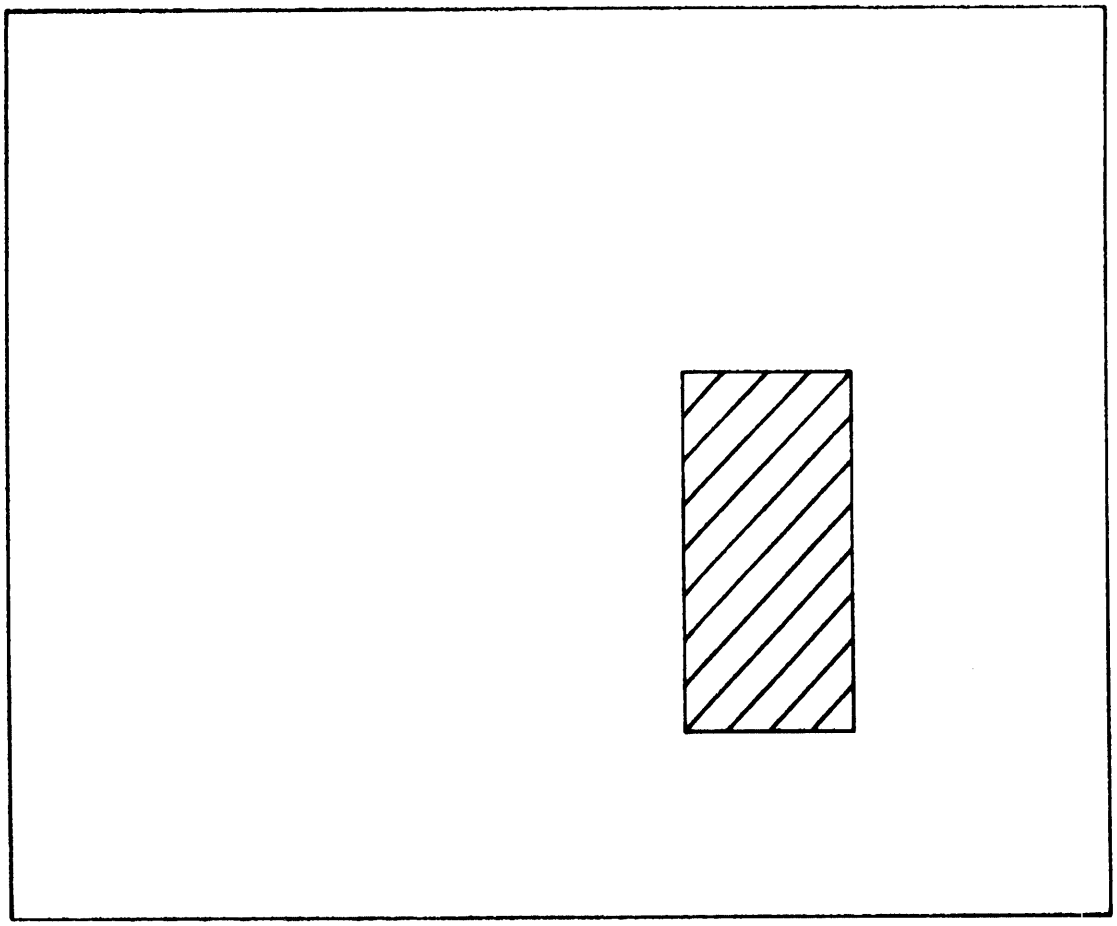
The secondary address 24, used in line 22, tells the display to "refresh" a character. The statement PRINT @32,24: "A" causes the character A to be written but not stored permanently on the display. Each time that statement is executed, the character "A" is refreshed on the display for about ¼ second. If the character "A" is to be observed for more than ¼ second, the statement PRINT @32,24; "A" must be executed repeatedly. Since line 22 is executed repeatedly, the refreshed arrow character remains steadily visible on the display but does not leave any trace when its position is changed. Only the first character of the string in the PRINT statement is refreshed. For example, the statement PRINT @32,24: "XYZ" will cause only the "X" character to be refreshed.

Any of the data editing functions described in Section 2 can be incorporated into a program of the type listed above. In order to keep this example as simple as possible, only the "change" function is implemented.

CROSS-HATCHING

Cross-hatching is a useful technique for highlighting data in certain applications. The clipping capability of the Graphic System makes this an easy task. A viewport corresponding to the area to be cross-hatched is set up. A FOR . . . NEXT loop is then executed, which covers the entire display with diagonal lines. The only lines which appear are those which fall inside the viewport. The lines which fall outside the viewport are clipped and do not appear. The example below shows how this is done.

```
950 INIT
960 DIM P(4,2)
970 VIEWPORT 0,130,0,100
980 WINDOW 0,1300,0,1000
990 MOVE 800,200
1000 INPUT @32,24:P(1,1),P(1,2)
1010 RDRAW 0,400
1020 INPUT @32,24:P(2,1),P(2,2)
1030 RDRAW 200,0
1040 INPUT @32,24:P(3,1),P(3,2)
1050 RDRAW 0,-400
1060 INPUT @32,24:P(4,1),P(4,2)
1070 RDRAW -200,0
1080 VIEWPORT P(1,1),P(3,1),P(1,2),P(2,2)
1090 MOVE 0,0
1100 SCALE 1,1
1110 FOR I=-100 TO 100 STEP 5
1120 MOVE 0,I
1130 DRAW 100,I+100
1140 NEXT I
1150 WINDOW 0,1300,0,1000
1160 VIEWPORT 0,130,0,100
1170 HOME
1180 END
```



This example fills a rectangle 200 units wide by 400 units high with diagonal lines. The lower left corner of the rectangle is located by the MOVE command in line 990. In the above example, it is specified to be (800,200). The example has five functional sections: initialization (lines 950 through 990), placing the actual display location (in GDU's) of each corner of the rectangle into each row of the array P (lines 1000 through 1080), setting up the proper scale and viewport for the diagonal lines (lines 1080 through 1100), drawing the diagonal lines (lines 1110 through 1140), and restoring the original window and viewport specifications. Each section is now discussed in more detail.

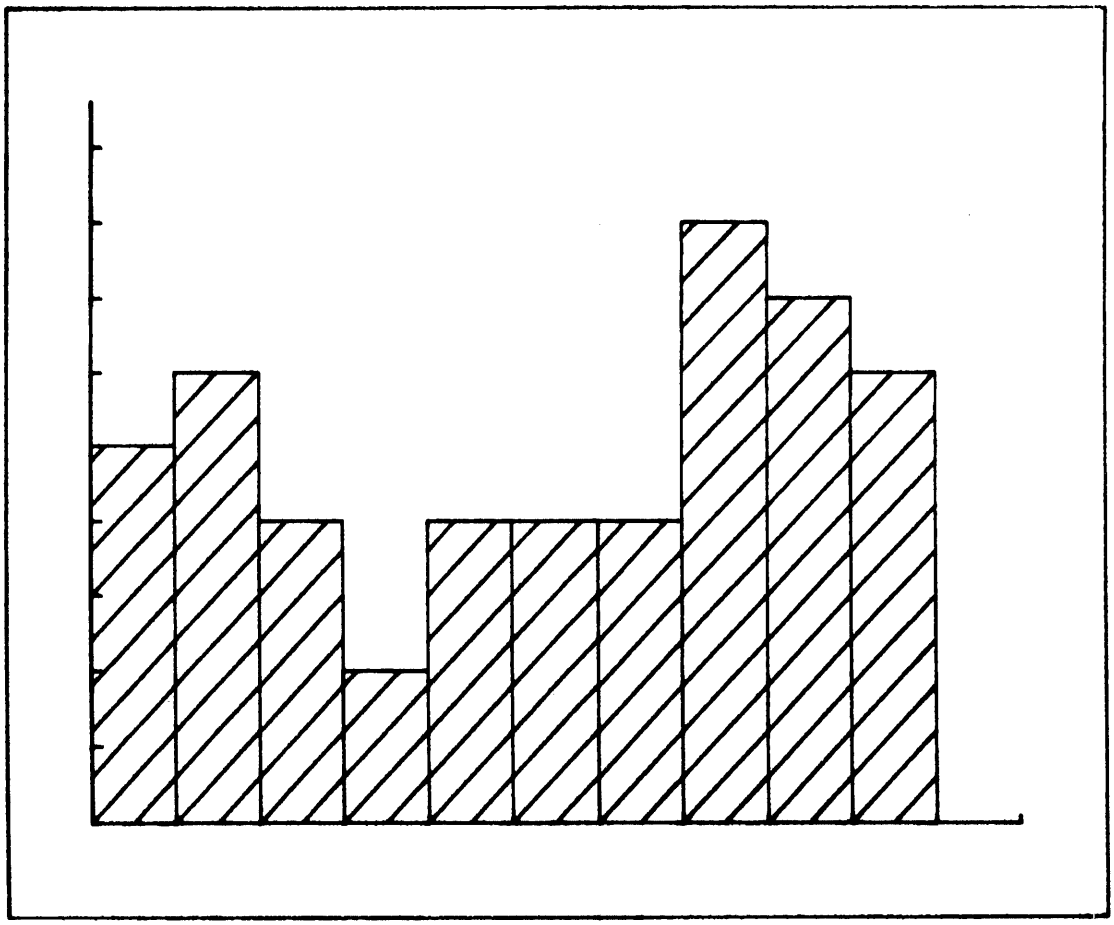
The first section dimensions array P, sets up the desired window and viewport, and locates the lower left corner of the rectangle by placing the graphic point there.

The second section draws the rectangle while storing the actual display location of each corner into a row of array P. The command used to determine the actual display location of the graphic point is INPUT @32,24: X,Y. This command is analogous to the GIN com-

mand. The major difference between the two commands is that GIN places the position of the graphic point in user data space into the two target variables. INPUT @32,24: X,Y places the actual display location of the graphic point in GDU's into the two target variables. Executing either of these commands is similar to asking "where is the pen?", except that the display and not a plotter is being addressed. GIN returns the locations of the graphic point in user data space; INPUT @32,24: returns the actual physical location of the graphic point in GDU's, the units used to designate specific locations on the display.

The third section sets up a viewport which coincides with the rectangle which has been drawn (line 1080). A SCALE command is executed to ensure the diagonal lines always appear at the same slope with the same distance between them, regardless of what viewport is defined. In this case, VIEWPORT's only function is to specify where the lines are to be clipped. Before a scale command can be executed, the graphic point must be moved to the place where the origin or (0,0) point of the new coordinate system is to be located. The MOVE 0,0 at line 1090 does this. It places the graphic point at (0,0), the lower left corner of both the window and viewport. However, because of the revised VIEWPORT command line 1080, the point (0,0) is now at the lower left corner of the box drawn on the display. (This can be observed by temporarily inserting the following statement into the program: 1095 END. When the program stops execution just after line 1090, the flashing cursor will appear at the lower left corner of the box drawn on the display, not at the lower left corner of the whole display.) When the SCALE command is executed, the new origin of user data space, i.e., the point (0,0), will be located at the lower left corner of the box drawn on the display. The diagonal lines drawn by the FOR . . . NEXT loop at lines 1110 through 1140 are drawn identically, regardless of the size or location of the box. Statements 1150 and 1160 restore the window and viewport to their original specifications.

The next example program, listed below, is identical to the histogram program in Section 3.



```
100 INIT
110 PAGE
120 DIM A(10),P(4,2)
130 RESTORE
140 DATA 10,120,10,90
150 READ V1,V2,V3,V4
160 A=0
170 M=0
180 FOR J=1 TO 50
190 R=RND(-2)
200 I=INT(R*10+1)
210 A(I)=A(I)+1
220 M=M MAX A(I)
230 NEXT J
240 VIEWPORT V1,V2,V3,V4
250 W1=0
260 W2=11
270 W3=0
280 W4=M*1.2
290 WINDOW W1,W2,W3,W4
```

```
300 AXIS 1,1
310 FOR K=1 TO 10
320 MOVE K-1,0
330 GOSUB 1000
340 NEXT K
350 HOME
360 END


1000 INPUT @32,24:P(1,1),P(1,2)
1010 RDRAW 0,A(K)
1020 INPUT @32,24:P(2,1),P(2,2)
1030 RDRAW 1,0
1040 INPUT @32,24:P(3,1),P(3,2)
1050 RDRAW 0,-A(K)
1060 INPUT @32,24:P(4,1),P(4,2)
1070 RDRAW -1,0
1080 VIEWPORT P(1,1),P(3,1),P(1,2),P(2,2)
1090 MOVE W1,W3
1100 SCALE 1,1
1110 FOR I=-100 TO 100 STEP 5
1120 MOVE 0,I
1130 DRAW 100,I+100
1140 NEXT I
1150 WINDOW W1,W2,W3,W4
1160 VIEWPORT V1,V2,V3,V4
1170 MOVE 0,A(K)
1180 RETURN
```

The major change is that the vertical bars which comprise the histogram are cross-hatched with diagonal lines. The subroutine beginning at line 1000 is the same program which was just discussed, except that the location and height of each vertical bar is determined by a data value in array A. This shows that a cross-hatching capability is easily added to a program by including it in a subroutine.

Section 8

PICTURES

IMPLICATIONS OF SCALE

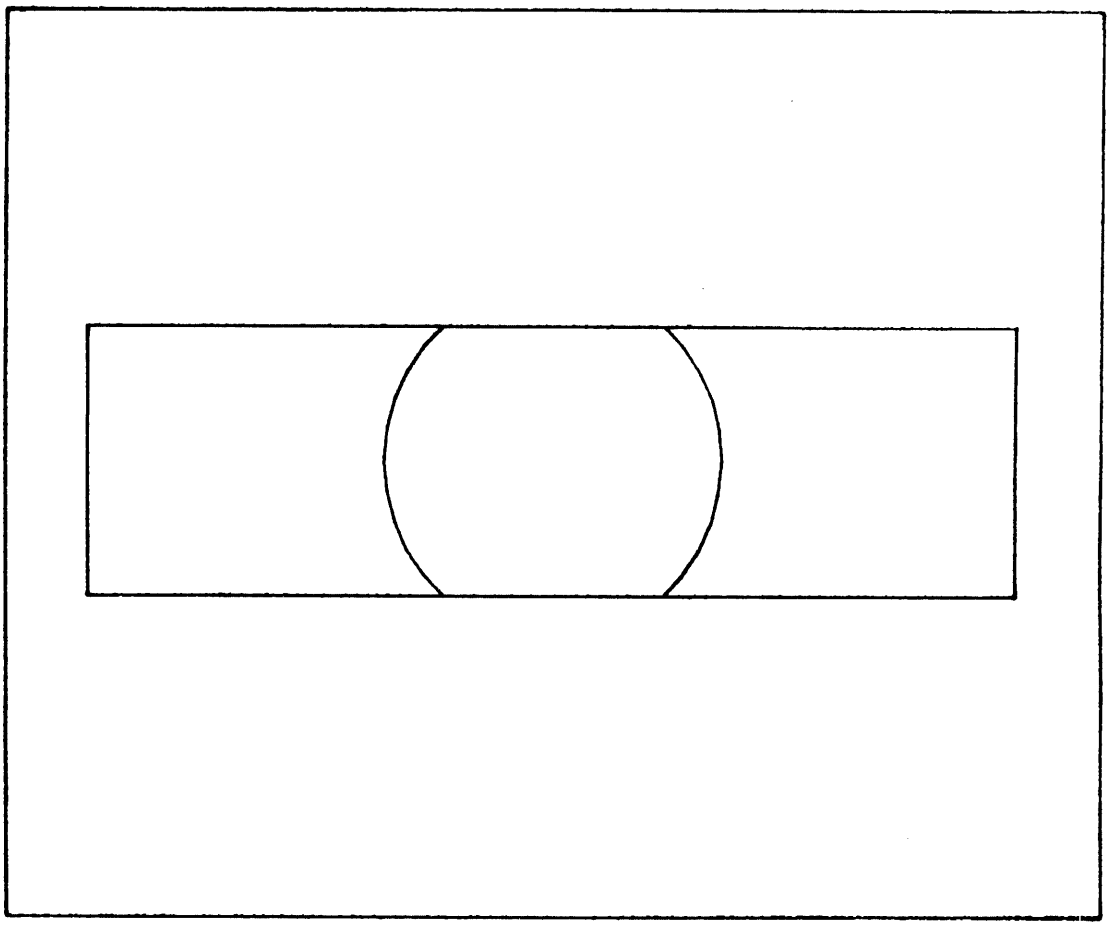
Drawing pictures with the GS is slightly different from representing data. When drawing a picture, it is important that the horizontal and vertical scale be equal. If they are not, objectionable distortion results. The aspect ratio of a rectangle is the quotient of its width and height. Whenever the aspect ratios of the window and viewport differ, the implied scaling factors, horizontal and vertical, will differ also. This was illustrated in Section 1. When drawing a picture, the limits in the WINDOW command must be changed whenever the viewport aspect ratio changes. If the SCALE command is used, the viewport determines only where the picture will be drawn on the display and what the clipping limits will be. With SCALE, the defined viewport has no direct influence on what the scale factors are. The following program serves as an illustration of this characteristic.

```
100 INIT
110 PAGE
120 RESTORE
130 DATA 0,130,0,100
140 READ V1,V2,V3,V4
150 VIEWPORT V1,V2,V3,V4
160 REM  DRAW A BOX AROUND THE VIEWPORT
170 AXIS
180 AXIS 0,0,130,100
190 REM  CENTER GRAPHIC POINT IN VIEWPORT
200 MOVE 65,50
210 REM  SPECIFY SCALE (DATA UNITS PER GDU)
220 SCALE 1,1
1000 REM  DRAW A CIRCLE OF RADIUS 20, CENTERED AT (0,0)
1010 SET DEGREES
1020 MOVE 20,0
1030 FOR I=10 TO 360 STEP 10
1040 DRAW 20*COS(I),20*SIN(I)
1050 NEXT I
1060 END
```

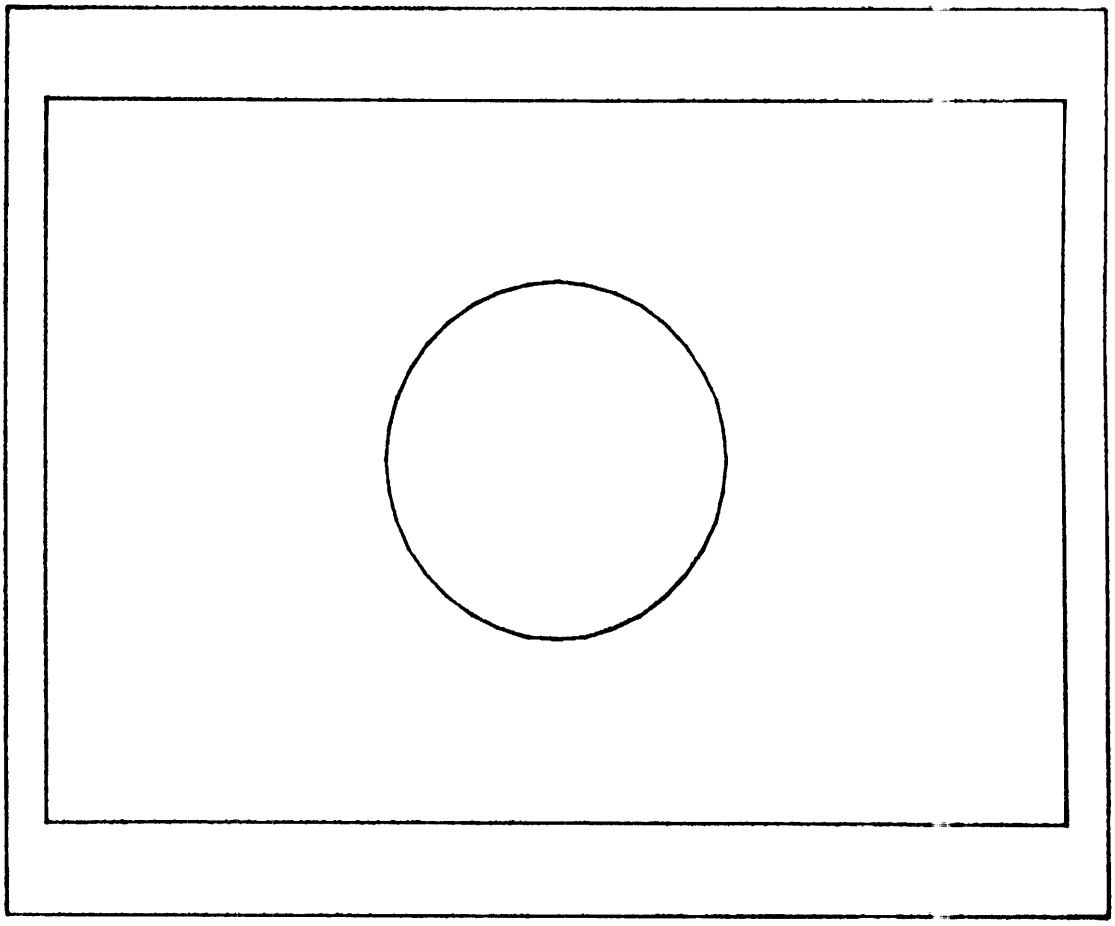
The INIT statement at line 100 sets up the default window and viewport. Statement 150 changes the viewport, but the window remains defined as it was in the INIT statement: an implied WINDOW 0,130,0,100. The two AXIS statements draw a box around the viewport so that its limits may be seen in the example programs. When the SCALE command is executed, the entire mapping transformation between user data space and screen space is redefined. The new origin, or (0,0) point of this coordinate system is defined to be the position of the graphic point when SCALE is executed. The clipping limits of the viewport, that is, the locations in user data space of the viewport edges, are therefore implied by the position of the graphic point relative to the viewport at the time the SCALE command is executed. The MOVE 65,50 command at line 200 thus becomes very important. Because it determines the graphic point's location when SCALE is executed, it also determines

what clipping limits will be implied by the SCALE statement. The SCALE command explicitly specifies scale factors and implicitly specifies clipping limits. In contrast, the WINDOW command explicitly specifies clipping limits and implicitly specifies scale factors. In all the sample runs which use this program (outputs shown below), the circle drawn by lines 1000 through 1060 is always centered in the viewport. This is the case regardless of what the viewport has been defined to be. Even though the viewport changes, the circle is always the same size and shape.

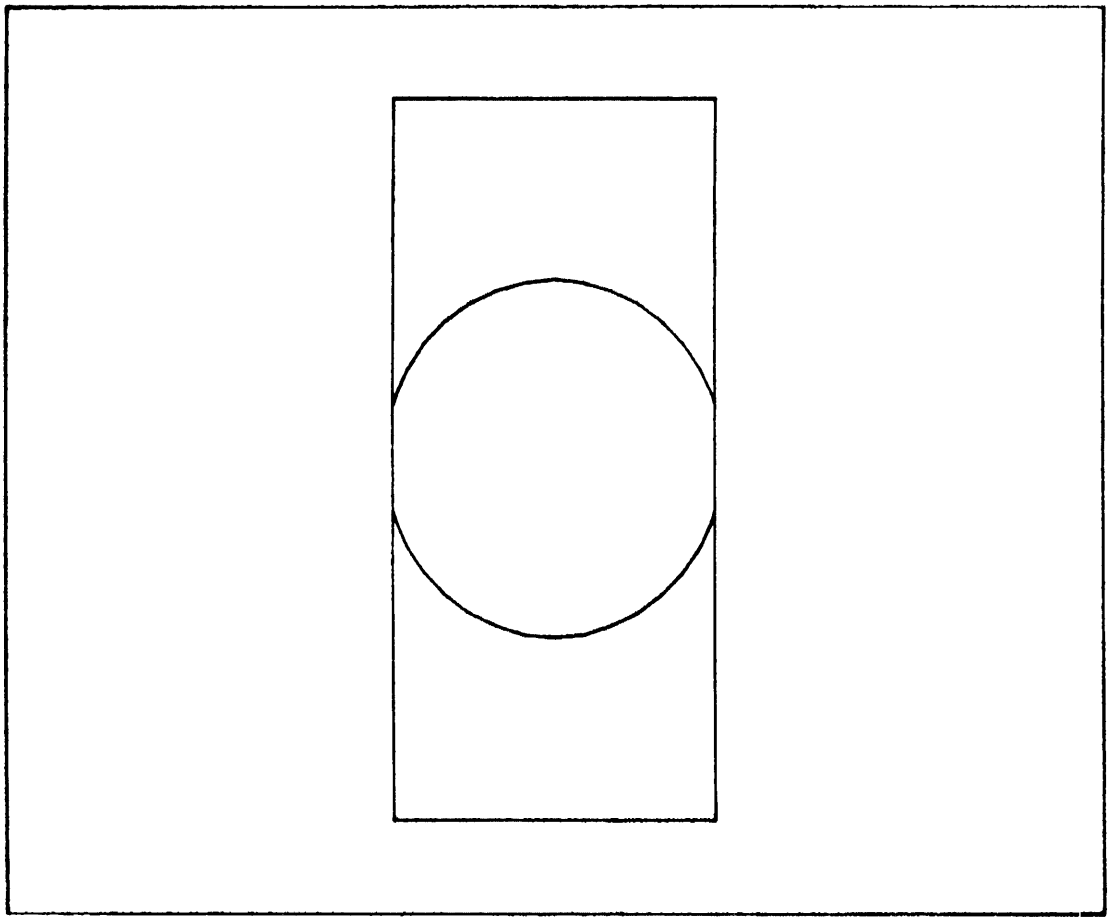
```
130 DATA 0,130,35,65
RUN
```



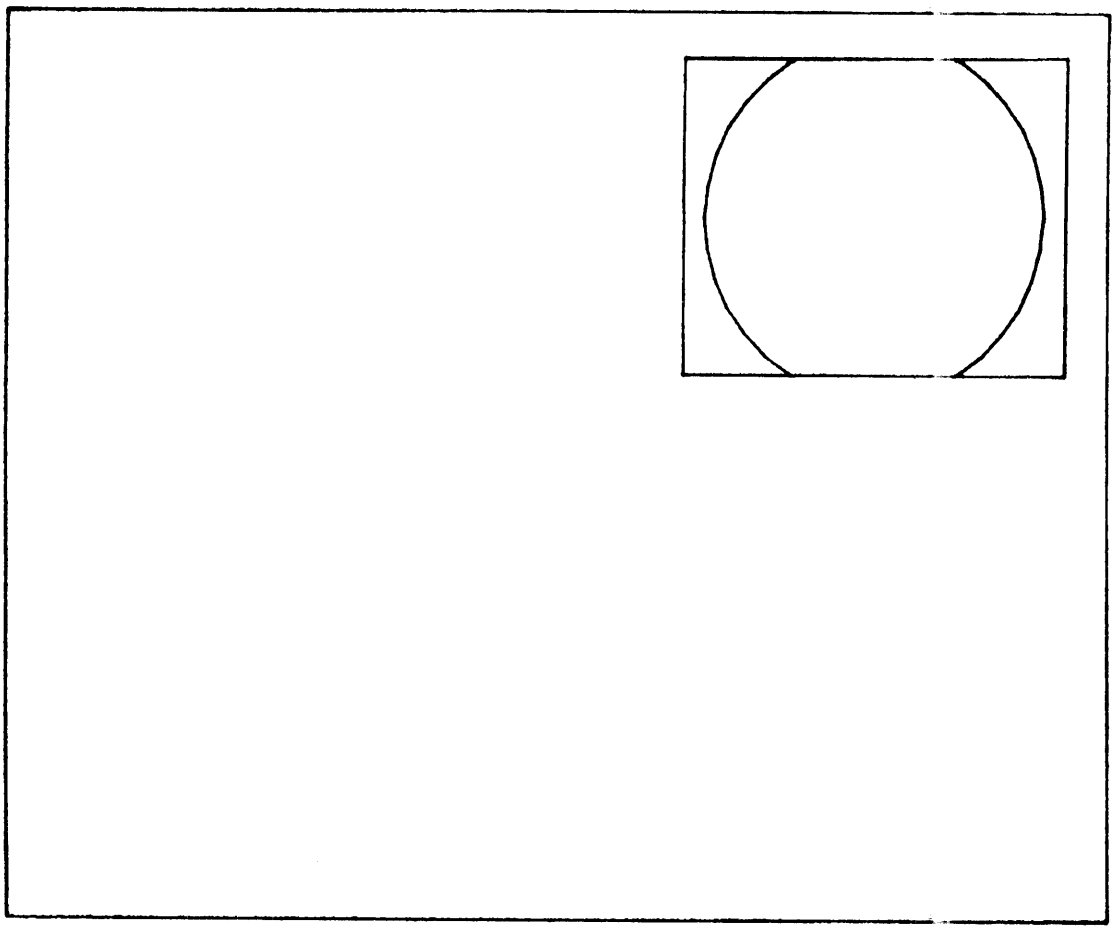
```
130 DATA 0,130,10,90
RUN
```



```
130 DATA 46,84,0,100
RUN
```



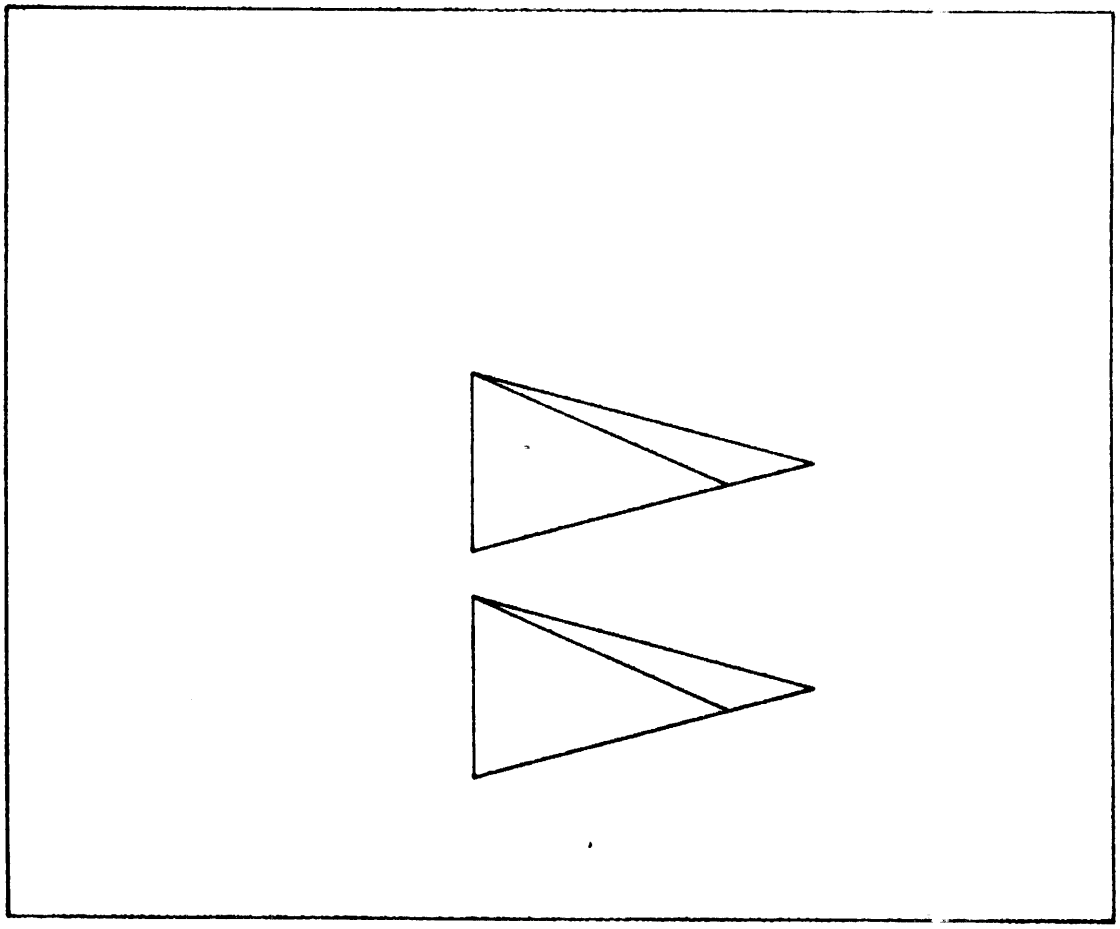
```
,

130 DATA 85,130,65,100
RUN
```



MANIPULATING OBJECTS

To define an object which is part of a picture, RDRAW and RMOVE commands are used most often. These commands permit the most flexibility when manipulating the resulting image. There are three ways an object or graphic entity is manipulated: changing its position, changing its orientation, and changing its size. If the object is drawn several times, the RDRAW and RMOVE commands which define it are placed in a subroutine. The size, orientation and location of the object can then be specified before the subroutine is called. The following example programs show this process.

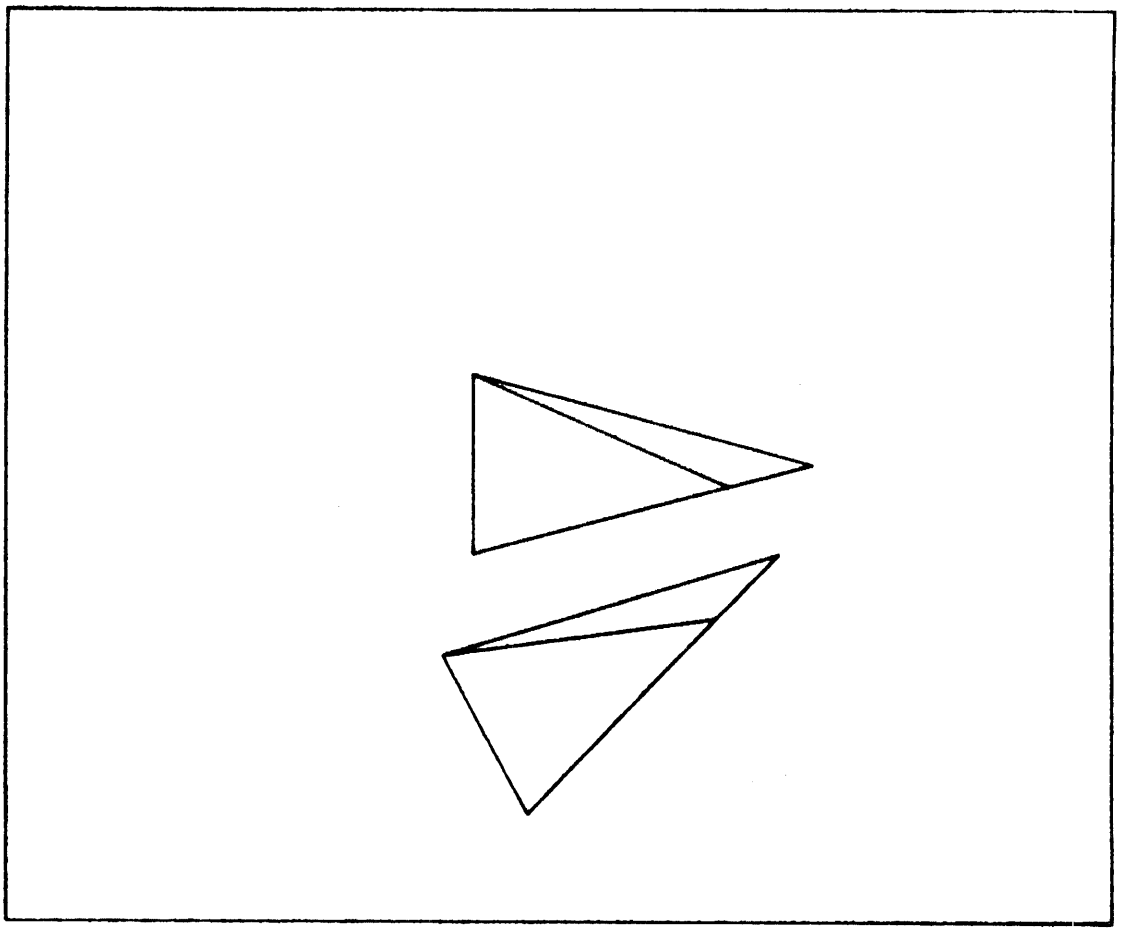
```
100 INIT
110 PAGE
120 DATA 0,130,0,100,0,130,0,100
130 READ V1,V2,V3,V4,W1,W2,W3,W4
140 VIEWPORT V1,V2,V3,V4
150 WINDOW W1,W2,W3,W4
160 SET DEGREES
170 ROTATE 0
180 MOVE 65,50
190 GOSUB 5000
200 MOVE 65,25
210 GOSUB 5000
220 HOME
230 END
5000 REM  DRAW TRIANGULAR FIGURE
5010 RMOVE -10,10
5020 RDRAW 0,-20
5030 RDRAW 40,10
5040 RDRAW -40,10
5050 RDRAW 30,-12.5
5060 RETURN
```



Beginning at line 5000 is a subroutine to draw the triangular figure and the extra line it contains. The MOVE command at statement 180 positions the graphic point in the center of the display. Control is passed to the subroutine at line 5000. After the subroutine draws the triangle and returns control, the MOVE at statement 200 is executed. This positions the graphic point at the lower center of the display. The subroutine at line 5000 is called again, drawing another triangle below the first one. In both cases, the location of the graphic point when line 5000 is executed determines where the triangle is drawn on the display.

The next example is functionally identical to the previous one except that the lower triangular figure is rotated.

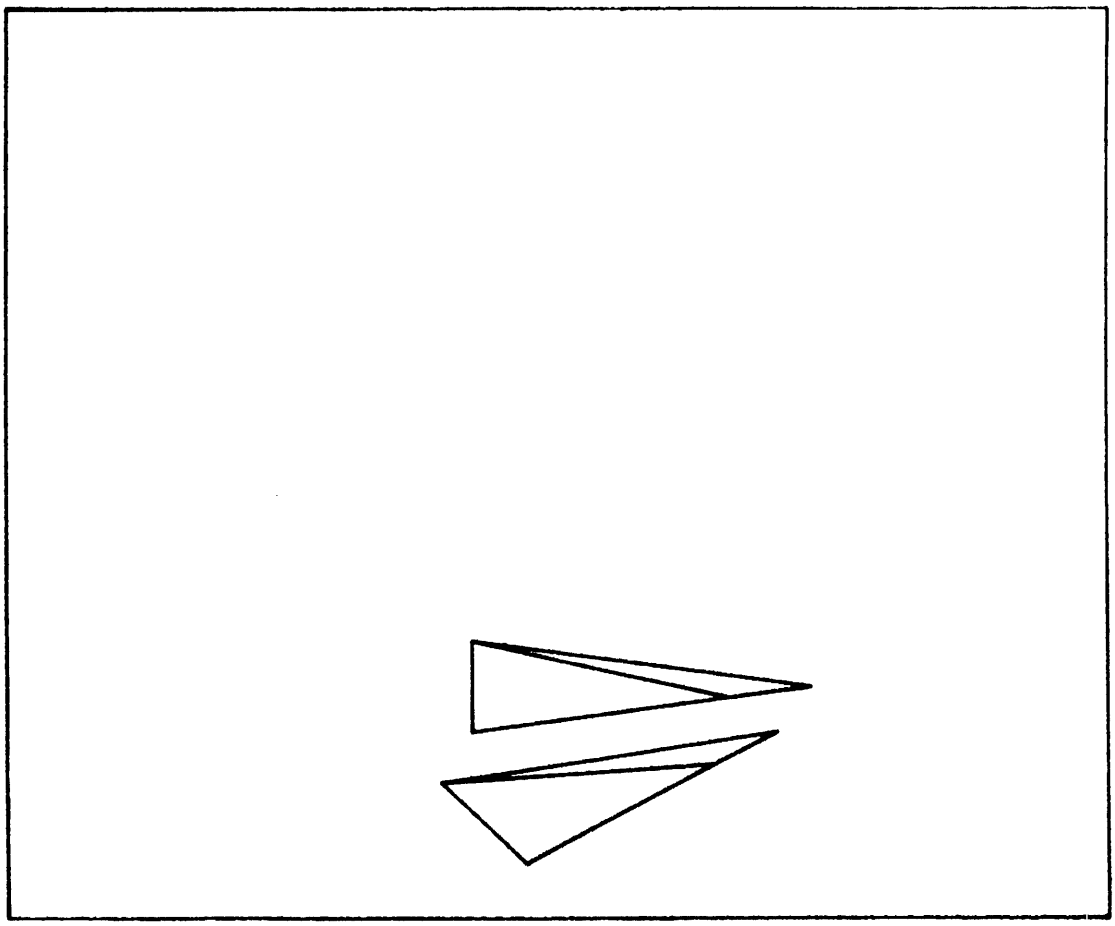
```
100 INIT
110 PAGE
120 DATA 0,130,0,100,0,130,0,100
130 READ V1,V2,V3,V4,W1,W2,W3,W4
140 VIEWPORT V1,V2,V3,V4
150 WINDOW W1,W2,W3,W4
160 SET DEGREES
170 ROTATE 0
180 MOVE 65,50
190 GOSUB 5000
200 MOVE 65,25
210 ROTATE 30
220 GOSUB 5000
230 HOME
240 END
5000 REM  DRAW TRIANGULAR FIGURE
5010 RMOVE -10,10
5020 RDRAW 0,-20
5030 RDRAW 40,10
5040 RDRAW -40,10
5050 RDRAW 30,-12.5
5060 RETURN
```



Inserted between the MOVE which positions the lower triangle (statement 200) and the GOSUB 5000 which actually draws the triangle (statement 220) is a ROTATE 30 statement. This rotates all subsequent RMOVEs and RDRAWs 30 degrees counterclockwise from their normal orientation. As a result, the second triangle is drawn rotated 30 degrees from its previous orientation.

The next example is the same as the previous one except that the window is changed. Its height is now 200 units, meaning that the aspect ratio of the window differs from the aspect ratio of the viewport. As a result, the horizontal and vertical scaling factors are no longer equal.

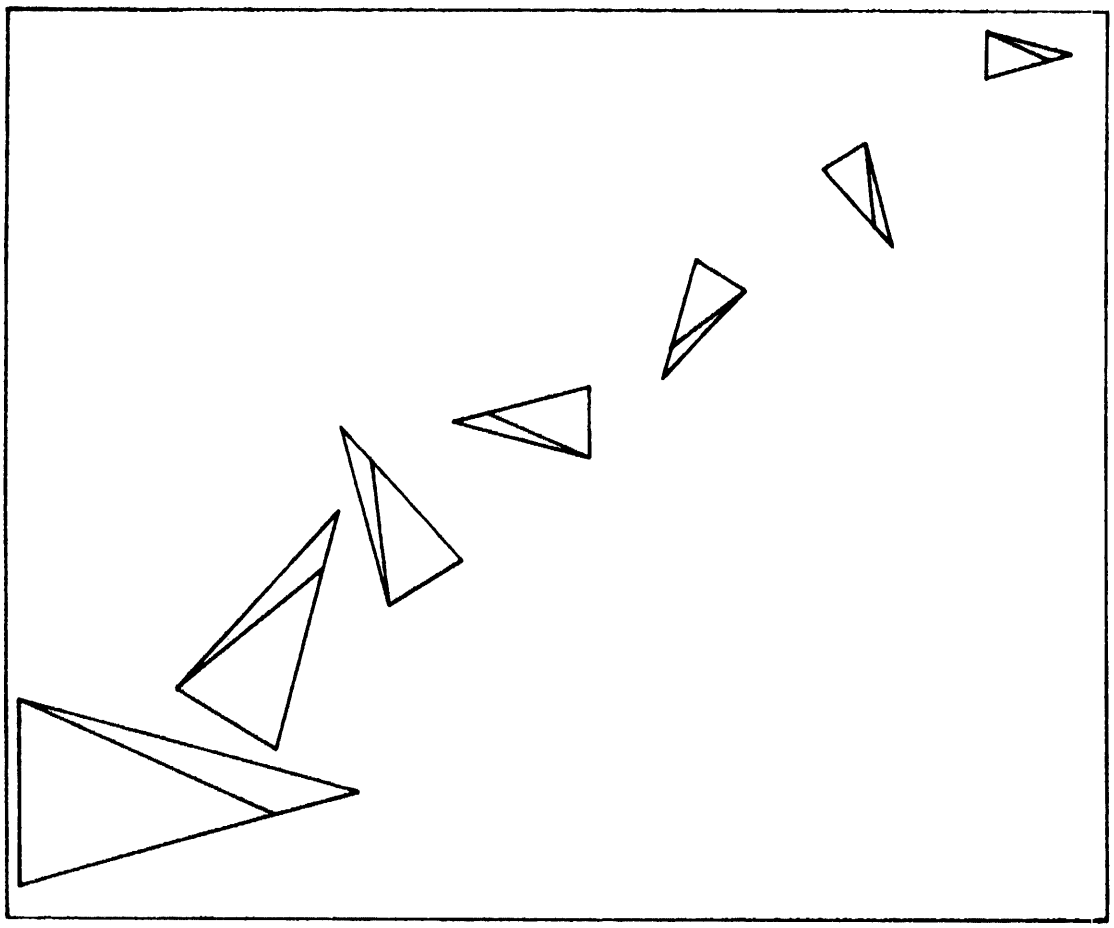
```
100 INIT
110 PAGE
120 DATA 0,130,0,100,0,130,0,200
130 READ V1,V2,V3,V4,W1,W2,W3,W4
140 VIEWPORT V1,V2,V3,V4
150 WINDOW W1,W2,W3,W4
160 SET DEGREES
170 ROTATE 0
180 MOVE 65,50
190 GOSUB 5000
200 MOVE 65,25
210 ROTATE 30
220 GOSUB 5000
230 HOME
240 END
5000 REM  DRAW TRIANGULAR FIGURE
5010 RMOVE -10,10
5020 RDRAW 0,-20
5030 RDRAW 40,10
5040 RDRAW -40,10
5050 RDRAW 30,-12.5
5060 RETURN
```



Although both triangles are distorted, the rotated triangle's shape is completely altered.

The next example combines positioning, rotation, and scaling. (The window parameters have been restored to their previous values. The horizontal and vertical scaling factors are again equal.)

```
100 INIT
110 PAGE
120 DATA 0,130,0,100
130 RESTORE
140 READ V1,V2,V3,V4
150 VIEWPORT V1,V2,V3,V4
160 SET DEGREES
170 FOR I=0 TO 90 STEP 15
180 WINDOW 0,110,0,110
190 MOVE 10+I,15+I
200 ROTATE I*4
210 SCALE 1+I/30,1+I/30
220 GOSUB 5000
230 NEXT I
240 HOME
250 END
5000 REM  DRAW TRIANGULAR FIGURE
5010 RMOVE -10,10
5020 RDRAW 0,-20
5030 RDRAW 40,10
5040 RDRAW -40,10
5050 RDRAW 30,-12.5
5060 RETURN
```



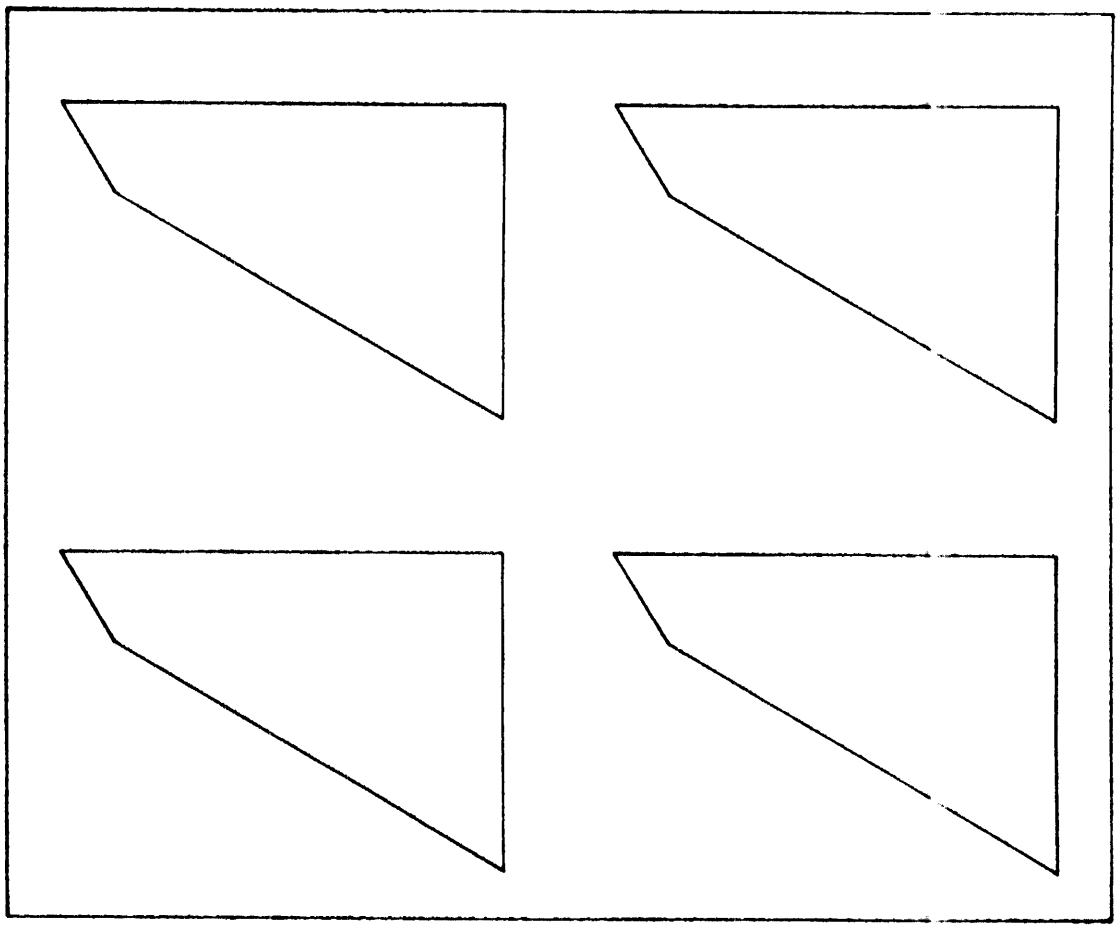
The heart of the example is the FOR . . . NEXT loop at statements 170 through 230. The WINDOW statement at line 180 restores a known window, ensuring that the MOVE command at statement 190 always places the graphic point consistently on the display. It is this statement which determines the position of each triangular figure on the display. The ROTATE I*4 command (at statement 200) specifies the orientation of each triangular figure. The SCALE 1+I/30,1+I/30 command (at statement 210) determines the size of each figure. Because of the MOVE, ROTATE, and SCALE commands, the same subroutine draws the triangular figure with seven different locations, orientations and sizes.

RDRAW and RMOVE do not intrinsically determine the position, orientation or size of the image they draw. For that reason, RDRAW and RMOVE are the most convenient graphic commands with which to define objects.

REVERSE VIEWPORT

The example below shows an irregular four sided figure drawn in each of the four quarters of the display.

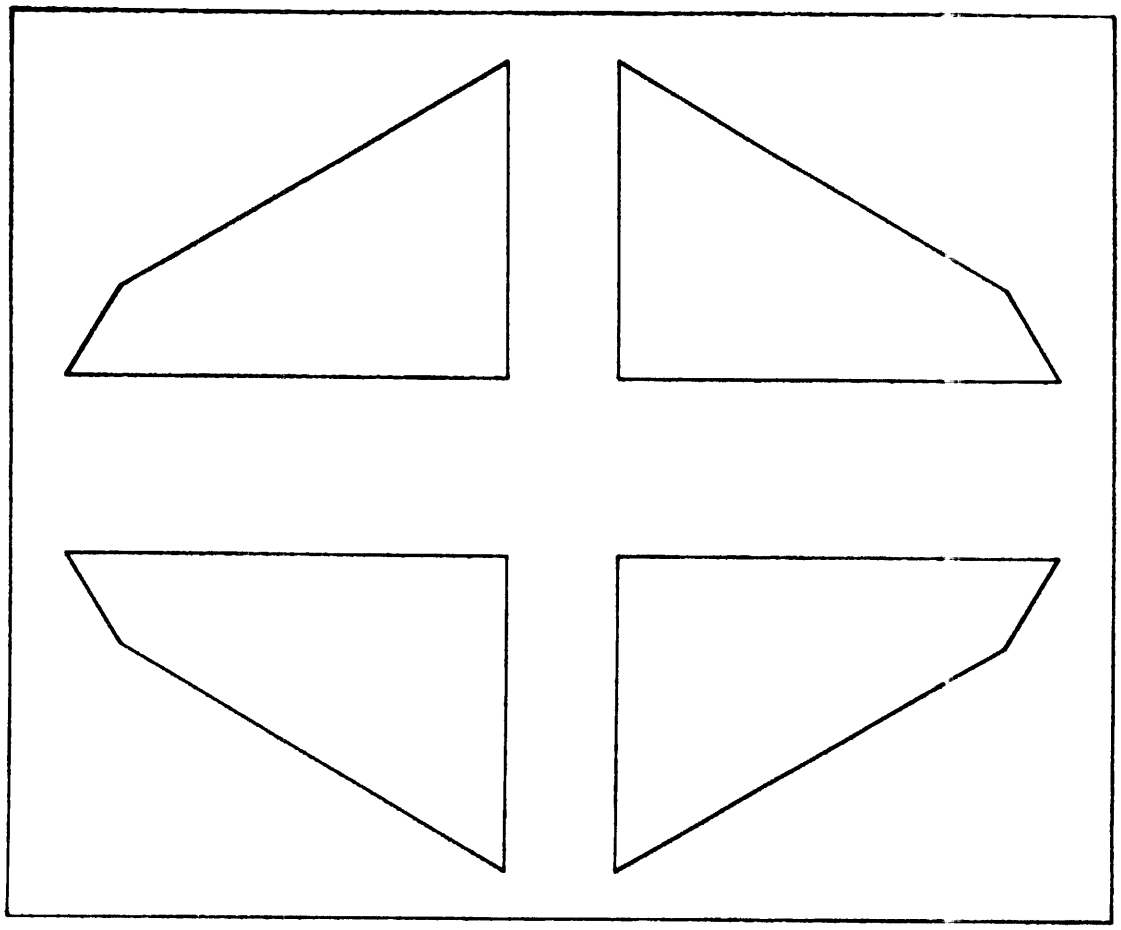
```
100 PAGE
110 INIT
120 WINDOW 0,100,0,50
130 VIEWPORT 0,65,0,50
140 GOSUB 1000
150 VIEWPORT 65,130,0,50
160 GOSUB 1000
170 VIEWPORT 0,65,50,100
180 GOSUB 1000
190 VIEWPORT 65,130,50,100
200 GOSUB 1000
210 HOME
220 END
1000 MOVE 90,40
1010 DRAW 10,40
1020 DRAW 20,30
1030 DRAW 90,5
1040 DRAW 90,40
1050 RETURN
```



The viewport is changed to alter the figure's position on the display. (This example is very different from other examples in this section. The image is drawn solely with absolute MOVE and DRAW commands. No RDRAWs and RMOVEs are used at all.)

The following example is identical to the previous one with three exceptions: the VIEWPORT statements at lines 150, 170, and 190.

```
100 PAGE
110 INIT
120 WINDOW 0,100,0,50
130 VIEWPORT 0,65,0,50
140 GOSUB 1000
150 VIEWPORT 130,65,0,50
160 GOSUB 1000
170 VIEWPORT 0,65,100,50
180 GOSUB 1000
190 VIEWPORT 130,65,100,50
200 GOSUB 1000
210 HOME
220 END
1000 MOVE 90,40
1010 DRAW 10,40
1020 DRAW 20,30
1030 DRAW 90,5
1040 DRAW 90,40
1050 RETURN
```



In line 150, the first two arguments of the VIEWPORT statement are reversed. In line 170, the second two arguments are reversed. In line 190, both pairs of arguments are reversed. The result in each case is a corresponding reversal of the displayed image. If the first two viewport arguments (which specify the horizontal location of the viewport) are reversed, the image is reversed horizontally. If the second two arguments (which specify the vertical locations of the viewport) are reversed, the image is reversed vertically.

This capability is useful in applications where the GS must draw symmetrical images.

Section 9

THREE DIMENSIONS

ANOTHER TRANSFORM

With a Cartesian coordinate system, specifying a point in space requires three coordinate values. Specifying a point on a plane, such as the display of the Graphic System, requires only two coordinate values. In order to draw three dimensional objects and surfaces on the GS, the three-value location for each point in space must be transformed into a two-value location on the GS display. In effect, the display serves as the mathematical analog of a pane of glass. The object or surface of interest is then examined "through" this viewplane.

This transformation process is very similar to the one described in Section 4, where applying a transform to data was categorized into two fundamental approaches. If the data in its raw form is of no interest, it is transformed when it is input and stored in its transformed state. In the second approach, the data is stored and edited in the GS in its original incoming state. It is transformed to a new state only when it is graphed.

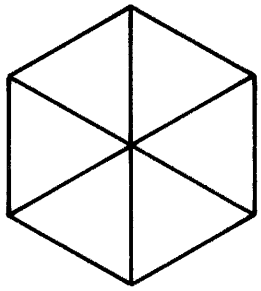
Three-dimensional transformations fall into the latter category. In several ways, three-dimensional transforms are similar to the polar transform in Section 4. Both are applied just before the data is graphed. In both transforms, the minimum and maximum of the raw data has no direct proportional relationship to the minimum and maximum of the data as displayed on the GS. In both cases, polar and three-dimensional, the minimum and maximum determination for display locations must reflect the transformation being used. Objectionable distortion is introduced in both cases if the horizontal and vertical scale factors are different. In other words, the aspect ratio of the window must match the aspect ratio of the viewport.

TRANSFORMATION LIMITATIONS

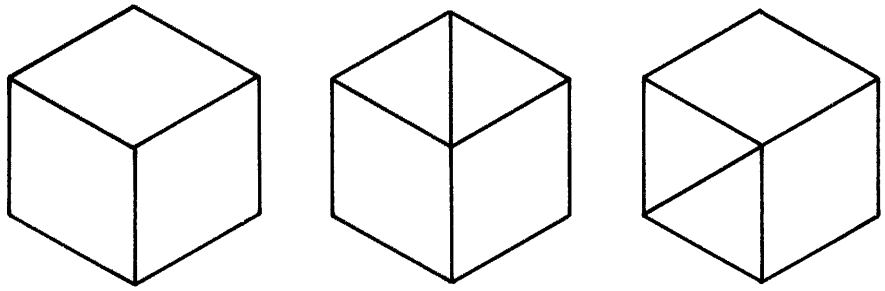
Conceptually, the GS display is a single plane, just as an artist's canvas or a photographic print is a plane. The process of representing a three dimensional object on any plane prevents true depth information about the object from being presented. This flatness creates ambiguities which can make the image on the display confusing. Some kind of stereo imaging device, which presents slightly different images to each eye of the viewer, is the only way to accurately convey this depth information.

If a three dimensional object is represented by a set of lines, as is usually the case with computer driven displays and plotters, there is another significant limitation. In certain circumstances, showing all the edges of the object introduces confusion.

What is the object shown below?



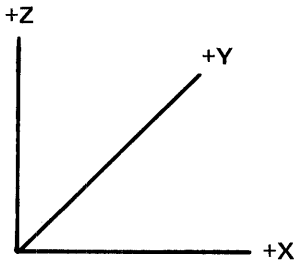
It could be any of the following:



There have been several algorithms written to remove the hidden lines which make the sketch of the cube so confusing. These algorithms all require sufficient topological information about the object to determine what lines to hide. While such algorithms are beyond the scope of this manual, they are discussed in works listed in the Reference Section.

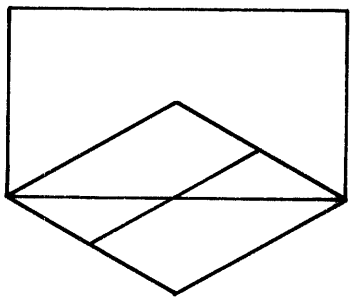
PROGRAMMING CONSIDERATIONS

The three dimensional coordinate system used in each example is the same.



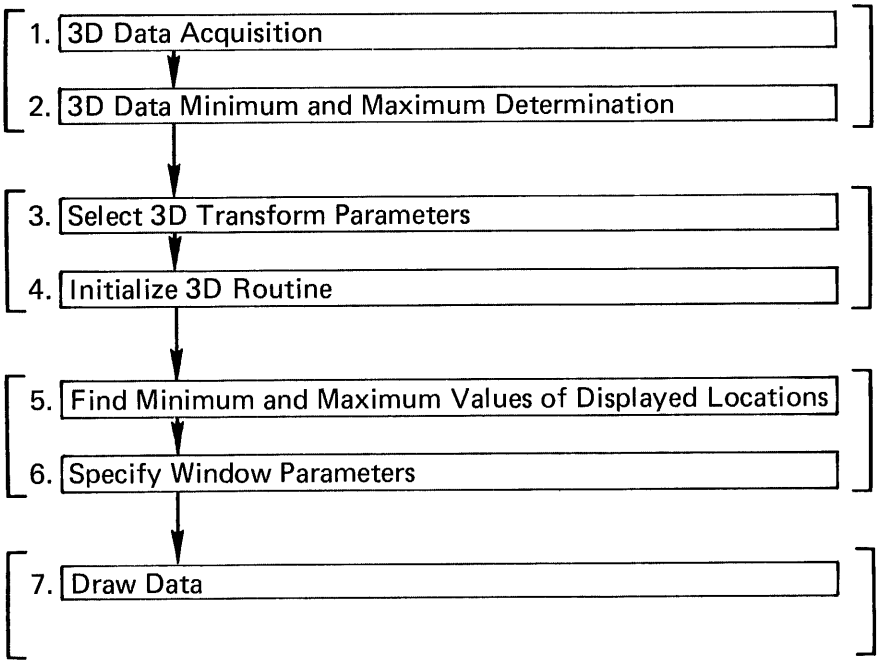
The positive X axis extends horizontally to the right of the origin. The positive Z axis extends vertically from the origin. The Y axis extends horizontally but "into" the paper (as depicted above). The positive Y axis is, in effect, the depth axis. This is a right-handed coordinate system.

Each example program in this section draws the same object, a variation on the unit cube. (The unit cube in standard position is a cube with one corner at the origin of the coordinate system, the (0,0,0) point. The diagonally opposite corner is at the point (1,1,1) in the coordinate system. All the edges of the cube are one distance unit in length.)



In all views shown of this object, the viewer is looking "down" upon it. It consists of a square drawn in the X-Y plane which has an additional line parallel to and 1/2 unit away from the Y axis. The four edges of this square are all one distance unit in length. There are four additional lines. These form a rectangle which has one corner at the origin or (0,0,0) point, and the diagonally opposite corner at the (1,1,1) point.

The example programs in this section draw three dimensional objects on the display with slightly different methods. However, each program can be separated into the same seven steps, shown below:

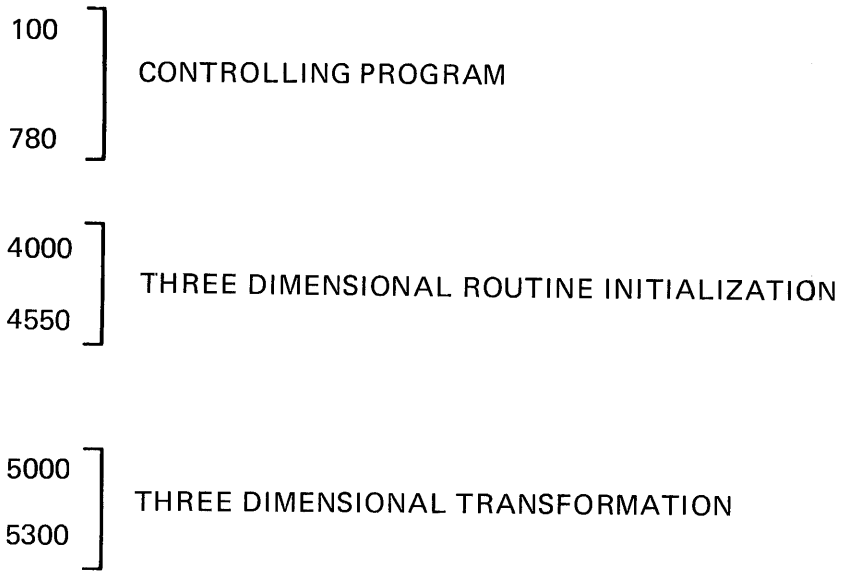


In the flow chart, the seven steps have been separated into four logical sections. Each of the three example programs in this part of the manual has the structure depicted above. In addition, each example program has another feature. The displayed image is automatically scaled and positioned such that it could not be any larger and still fit completely on the display. This function is performed by the two steps "find minimum and maximum of displayed locations", and "specify window parameters", listed above. If this automatic scaling feature is inappropriate for a particular application, it is easily removed.

The steps listed above are appropriately executed in the order shown. Furthermore, the paired steps logically go together. The first step in each pair derives or acquires some type of data.

The second step in each pair acts appropriately on that data. Examining the first pair of steps (above) provides an example. If the three dimensional data minimums and maximums have been determined at some point in the program's execution, there is no need to ever re-determine them unless the three dimensional data changes. Likewise, once the three dimensional routine is initialized, there is no need to re-initialize it unless there is a change in its parameters. However, if either the three dimensional data or the transform parameters change, the previously defined window may become inappropriate. Unless the window is intended to be constant, it should be re-defined.

Three methods of 3D to 2D transformations are described in this section. In all three methods, the structure and statement numbering of the sample programs are very similar.



The controlling program begins at statement 100 and never extends beyond statement 780. It has the structure shown in the seven step flow chart. The 3D routine initialization segments always begin at statement 4000 and never extend beyond statement 4550. The actual transformation routines always begin at statement 5000 and never extend beyond statement 5300. The array O always contains the object to be drawn. The array's size and contents are identical in all three examples.

Since the same object is drawn by each program, the statements which perform the "3D data acquisition" and "3D data minimum and maximum determination" functions are identical in all three examples. They are listed below.

```
100 INIT
110 DIM E(3),L(3),O(11,3),P(3),Q(2,3)
120 RESTORE
130 REM ******************
140 REM  FILL 3D DATA ARRAY
150 O=0
160 DATA 0.5,1,0.5,1,1
170 READ O(1,1),O(1,2),O(2,1),O(4,2),O(5,1)
180 DATA 1,1,1,1
190 READ O(5,2),O(6,1),O(8,3),O(9,1)
200 DATA 1,1,1,1
210 READ O(9,2),O(9,3),O(10,1),O(10,2)
220 REM  FIND MIN AND MAX OF 3D DATA
230 Q=-1.0E+300
240 DATA 1.0E+300,1.0E+300,1.0E+300
250 READ Q(1,1),Q(1,2),Q(1,3)
260 FOR I=1 TO 10
270 FOR J=1 TO 3
280 Q(1,J)=Q(1,J) MIN O(I,J)
290 Q(2,J)=Q(2,J) MAX O(I,J)
300 NEXT J
310 NEXT I
320 REM ******************
```

With the exception of the DIM command at statement 110, statements 100 through 320 of each example are identical. Statements 100 through 120 perform initialization. Statements 130 through 210 fill the data array O. Statements 220 through 310 determine the minimum and maximum of the data in each direction. These minimums and maximums are placed in array Q arranged as follows: row 1 of Q holds the minimums, row 2 of Q holds the maximums, column 1 of Q holds the X minimum and maximum, column 2 of Q holds the Y minimum and maximum, and column 3 of Q holds the Z minimum and maximum. In the three examples, the minimums are all 0 and the maximums are all 1.

ARRAY Q

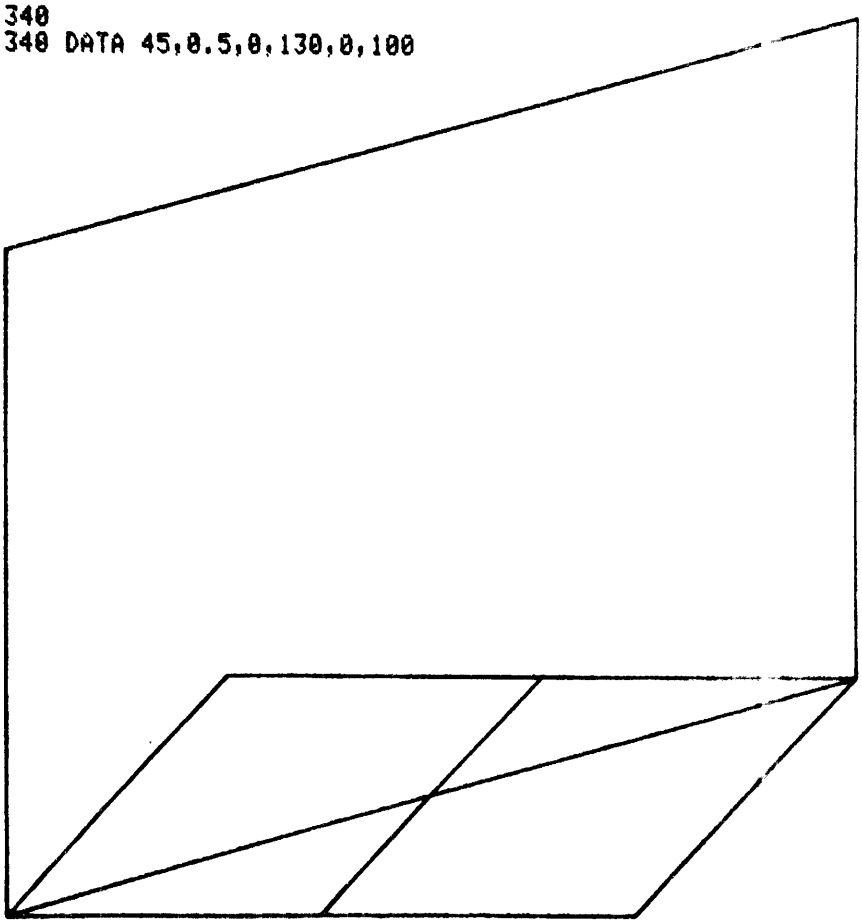
	COLUMNS 1	2	3
ROWS 1	X MINIMUM	Y MINIMUM	Z MINIMUM
2	X MAXIMUM	Y MAXIMUM	Z MAXIMUM

For example, the X maximum is contained in Q (2,1).

The individual 3D point to be transformed is passed to the transformation routine (at statement 5000) in the three-element array P. The 2D values to be displayed are returned to the controlling program in variables X and Y.

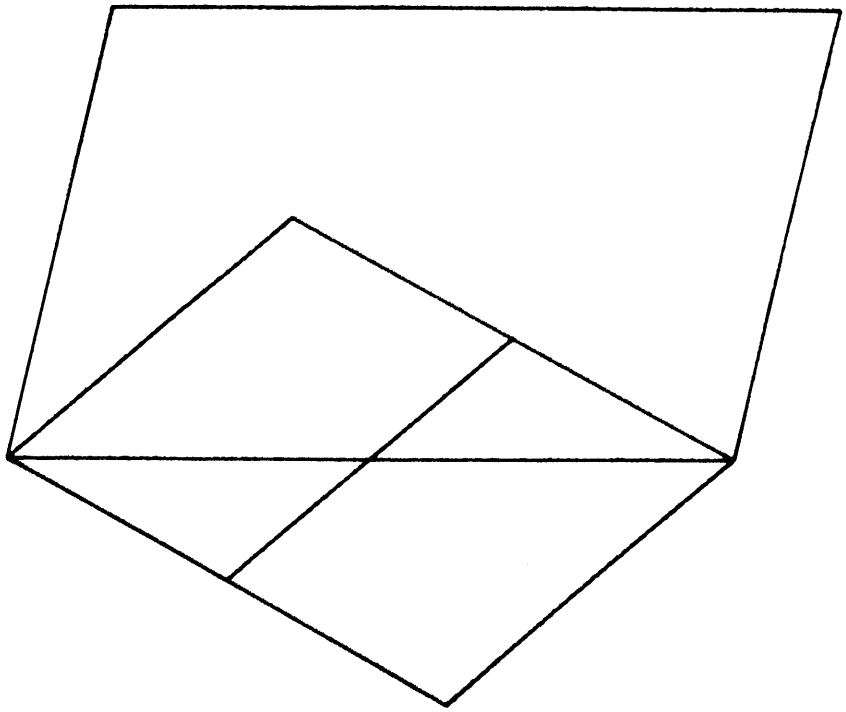
The transformation methods are discussed in increasing order of complexity.

```
340
340 DATA 45,0.5,0,130,0,100
```



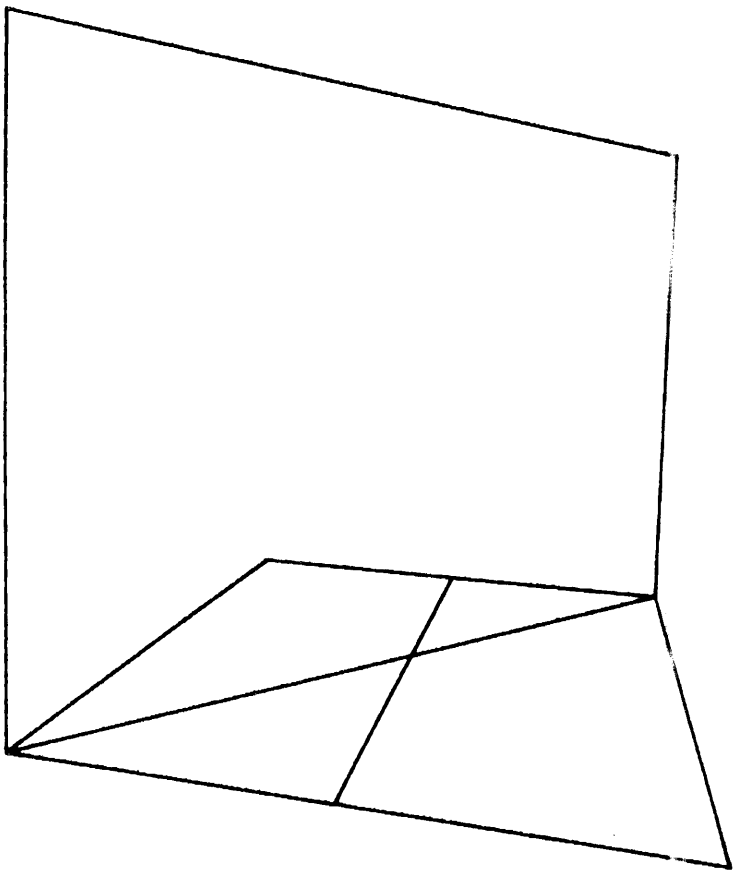
OBLIQUE PROJECTION

```
340 DATA 35.2,0,-25
```



ORTHOGRAPHIC PROJECTION

```
340 DATA 0.8,-1,0.6,0.5,0.5,0.5
```



PERSPECTIVE PROJECTION

The oblique transformation routine occupies the smallest amount of memory space, executes the most quickly, and introduces the most visual distortion. The perspective transformation routine occupies the largest amount of memory space, executes the most slowly, and introduces the least distortion. In all three criteria, the orthographic transformation routine falls in between the other two.

OBLIQUE PROJECTION

As used here, the term oblique transformation describes a class of transformations commonly used in certain technical drawing applications and variously called "oblique", "cabinet", and "cavalier". This class might appropriately be called "shear" transformations because of the distortion they introduce. Lines parallel to the X and Z axes are projected onto the viewplane with no distortion. Lines parallel to the Y axis, when projected onto the viewplane, become lines drawn at an angle to horizontal in the viewplane. This angle is the projection angle, variable A in the program. It is typically not less than 30 degrees and not greater than 45 degrees. The lengths of lines parallel to the Y axis are multiplied by the foreshortening factor prior to being drawn. This foreshortening factor is never greater than 1 and is usually not less than ½. With a foreshortening factor of ½, lines parallel to the Y axis are drawn half the length they would be drawn if the foreshortening factor were equal to 1.

In technical drawing terminology, the cavalier projection has a projection angle of 45 degrees and a foreshortening factor of 1, the cabinet projection has a projection angle of 45 degrees and a foreshortening factor of ½, and the oblique projection has a projection angle of 30 degrees and a foreshortening factor of 1. In this manual, the term oblique is used in a more general sense. All surfaces of the object which are parallel to the X-Z plane are drawn undistorted. All surfaces of the object parallel to the X-Y and Y-Z planes are distorted in varying amounts. This inherent distortion can be reduced for certain applications by specifying a projection angle of greater than 45 degrees and a foreshortening factor less that ½.

The entire oblique projection program is as follows:

```
100 INIT
110 DIM E(3),L(3),O(11,3),P(3),Q(2,3)
120 RESTORE
130 REM ****************
140 REM  FILL 3D DATA ARRAY
150 O=0
160 DATA 0.5,1,0.5,1,1
170 READ O(1,1),O(1,2),O(2,1),O(4,2),O(5,1)
180 DATA 1,1,1,1
190 READ O(5,2),O(6,1),O(8,3),O(9,1)
200 DATA 1,1,1,1
210 READ O(9,2),O(9,3),O(10,1),O(10,2)
220 REM  FIND MIN AND MAX OF 3D DATA
230 Q=-1.0E+300
240 DATA 1.0E+300,1.0E+300,1.0E+300
250 READ Q(1,1),Q(1,2),Q(1,3)
260 FOR I=1 TO 10
270 FOR J=1 TO 3
280 Q(1,J)=Q(1,J) MIN O(I,J)
290 Q(2,J)=Q(2,J) MAX O(I,J)
300 NEXT J
310 NEXT I
320 REM ****************
330 REM  SPECIFY 3D PARAMETERS - PROJ ANGLE AND FORESHORTENING FACTOR
340 DATA 45,0.5,0,130,0,100
350 READ A,F,W1,W2,W3,W4
360 REM  INITIALIZE 3D  ROUTINE
```

```
370 GOSUB 4000
380 REM ******************
390 REM SET UP VIEWPORT AND 2D WINDOW
400 VIEWPORT 0,130,0,100
410 WINDOW W1,W2,W3,W4
420 REM ********************
430 REM  MOVE TO FIRST POINT
440 P(1)=Q(1,1)
450 P(2)=Q(1,2)
460 P(3)=Q(1,3)
470 GOSUB 5000
480 MOVE X,Y
490 REM  DRAW OBJECT
500 FOR I=2 TO 11
510 P(1)=Q(I,1)
520 P(2)=Q(I,2)
530 P(3)=Q(I,3)
540 GOSUB 5000
550 DRAW X,Y
560 NEXT I
570 HOME
580 END
4000 REM  FIND SCALING FACTOR
4010 SET DEGREES
4020 REM  FIND WHETHER HORIZ OR VERT IS LARGEST SIZE OF IMAGE
4030 REM  H=HORIZONTAL IMAGE SIZE, V=VERTICAL IMAGE SIZE
4040 H=Q(2,1)-Q(1,1)+(Q(2,2)-Q(1,2))*F*COS(A)
4050 V=Q(2,3)-Q(1,3)+(Q(2,2)-Q(1,2))*F*SIN(A)
4060 IF H/V>1.3 THEN 4100
4070 REM  VERTICAL LARGER
4080 S=(W4-W3)/V
4090 GO TO 4120
4100 REM  HORIZONTAL LARGER
4110 S=(W2-W1)/H
4120 H=S*F*COS(A)
4130 V=S*F*SIN(A)
4140 RETURN
5000 REM TRANSFORM FROM DATA X,Y,Z TO SCREEN X,Y
5010 X=W1+P(1)*S+P(2)*H
5020 Y=W3+P(3)*S+P(2)*V
5030 RETURN
```

Statements 330 through 370 initialize the projection angle A, the foreshortening factor F and the window parameters. The initializing routine starting at statement 4000 computes the scale factor for the image. The computation of the scaling factor is based on the assumption that anything drawn on the display should be as large as possible within the specified window. The initialization section determines whether the limiting factor on the size of the displayed image is the viewport height or viewport width. The IF command at statement 4060 is the decision point. The scale factor is calculated on that basis. The remaining two statements in the section (4120 and 4130) merely calculate constants which allow faster execution of the two transformation statements (5010 and 5020).

When using oblique projection, the minimums and maximums in the display's coordinate system do not need to be computed. They are specified in advance. This is the oblique projection's principal advantage over the orthographic and perspective transformations. The initialization and transformation statements ensure that no data will be drawn outside the specified window. As a result, defining the window requires one WINDOW command and nothing else. Of course, the window height and width must have the same ratio as the viewport height and width if distortion is to be minimized.

Statements 430 through 480 transform the first point in array 0 and position the graphic point there with a MOVE command.

The remainder of the object is drawn with the FOR . . . NEXT loop at statements 500 through 560. As with the other examples in this section, the routine required to transform three data points into two data points for graphing begins at statement 5000.

WINDOW PARAMETERS

As stated previously, the aspect ratio of the viewport must match the aspect ratio of the window. This requirement complicates the initialization of the parameters used by the WINDOW command in the orthographic and perspective transformations (statements 390 through 610 in both examples).

```
380 REM ******************
390 REM  FIND MIN AND MAX IN VIEWPLANE'S COORDINATE SYSTEM
400 DATA 1.0E+300,-1.0E+300,1.0E+300,-1.0E+300
410 READ X1,X2,Y1,Y2
420 FOR I=1 TO 2
430 FOR J=1 TO 2
440 FOR K=1 TO 2
450 P(1)=Q(I,1)
460 P(2)=Q(J,2)
470 P(3)=Q(K,3)
480 GOSUB 5000
490 X1=X1 MIN X
500 X2=X2 MAX X
510 Y1=Y1 MIN Y
520 Y2=Y2 MAX Y
530 NEXT K
540 NEXT J
550 NEXT I
560 REM SET UP VIEWPORT AND 2D WINDOW
570 VIEWPORT 0,130,0,100
580 IF (X2-X1)/(Y2-Y1)>1.3 THEN 610
590 WINDOW (X2+X1)*0.5-0.65*(Y2-Y1),(X2+X1)*0.5+0.65*(Y2-Y1),Y1,Y2
600 GO TO 630
610 WINDOW X1,X2,(Y2+Y1)*0.5-(X2-X1)/2.6,(Y2+Y1)*0.5+(X2-X1)/2.6
620 REM ********************
```

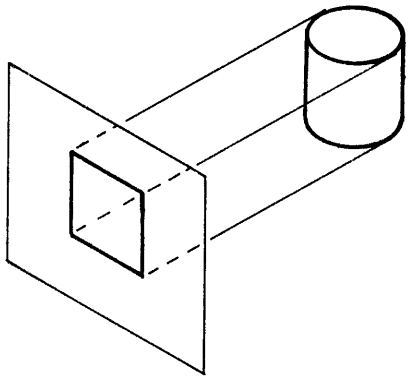
Since there is no simple way to determine where on the display the transformed image will appear before the three dimensional transformations are initialized, the window minimums and maximums must be computed from the three dimensional minimums and maximums after the transform is initialized. This is precisely the function performed by lines 390 through 610. Each corner of the cube created by the three dimensional minimums and maximums, a total of eight points in three dimensional space, is transformed into two dimensional coordinates. The window parameters are then derived from the minimums and maximums of these two dimensional values. The parameters are defined (in the two WINDOW commands in statements 590 and 610) such that the image on the display is always centered in the viewport.

The remainder of the main line program (statements 630 through 760) simply moves the graphic point to the location of the first transformed point and draws the object.

```
620 REM **********************
630 REM  MOVE TO FIRST POINT
640 P(1)=O(1,1)
650 P(2)=O(1,2)
660 P(3)=O(1,3)
670 GOSUB 5000
680 MOVE X,Y
690 REM  DRAW OBJECT
700 FOR I=2 TO 11
710 P(1)=O(I,1)
720 P(2)=O(I,2)
730 P(3)=O(I,3)
740 GOSUB 5000
750 DRAW X,Y
760 NEXT I
770 HOME
780 END
```

ORTHOGRAPHIC PROJECTION

Orthographic projection has less inherent distortion than oblique projection. In orthographic projection, a straight line connecting a point on the object and the projection of this point on the viewplane is always perpendicular to the viewplane.



As a result of this, all lines connecting points on the object to the corresponding points on the viewplane are parallel. In an actual three dimensional situation, this could happen only if the observer were an infinite distance away from the object being viewed. Having all the lines of projection parallel introduces ambiguities of perspective similar to those which occur when viewing an airplane silhouette at great distance. In that situation, it is often easy to recognize the silhouette as an airplane but difficult to tell whether it is coming toward you or going away.

Listed below are the statements which comprise the heart of the orthographic example.

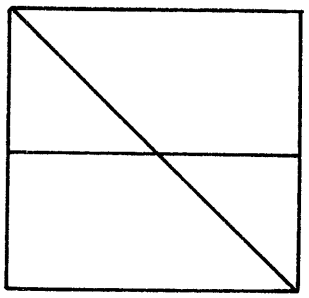
```
330 REM  SPECIFY 3D PARAMETERS - THREE  ROTATION ANGLES
340 DATA 35.2,0,25
350 READ A1,A2,A3

4000 SET DEGREES
4010 C1=COS(A1)
4020 S1=SIN(A1)
4030 C2=COS(A2)
4040 S2=SIN(A2)
4050 C3=COS(A3)
4060 S3=SIN(A3)
4070 RETURN

5000 REM TRANSFORM P(1),P(2),P(3) INTO X,Y
5010 X=C2*C3*P(1)-C2*S3*P(2)+S2*P(3)
5020 Y=(S1*S3-C1*S2*C3)*P(1)+(S1*C3+C1*S2*S3)*P(2)+C1*C2*P(3)
5030 RETURN
```

The initialization section (beginning at statement 4000) requires three parameters. They are rotation angles which specify the object's orientation when viewed: a rotation angle around the X axis (A1), a rotation angle around the Y axis (A2), and a rotation angle around the Z axis (A3). The rotation specified is in a consistent direction around each axis. If the thumb of the right hand is pointed in the positive direction of the axis to be rotated around, the fingers curl in the direction of a positive angular rotation. Inherent in the transformation section (beginning at statement 5000) is the order in which the rotations are performed: first around the Z axis, then around the Y axis, and finally around the X axis. This ordering is fixed in the transformation equations and has an important characteristic: it is not commutative. A rotation around X followed by a rotation around Y produces a different orientation than a rotation around Y followed by a rotation around X. The ordering used in this example (first Z, then Y, then X) is one of six ways to specify the order in which the rotations occur.

When the orientation angles are all zero, the object's coordinate system is oriented as described earlier in this section. The 3D Z axis is parallel to the display's vertical axis (with the positive direction being up), the 3D X axis is parallel to the display's horizontal axis (with the positive direction being to the right), and the 3D Y axis is perpendicular to the display's surface (with the positive direction being "into" the display).

When rotation angles are specified, they occur relative to this coordinate system which always remains fixed, regardless of the specified rotation angles. An example is the case of a rotation around the Z axis followed by a rotation around the X axis. Regardless of the object's orientation after its rotation around the Z axis, the X axis it is then rotated around will always be parallel to a horizontal line on the display's surface, not to the rotated X axis of the object's coordinate system.

For example, the object is rotated 90 degrees around the Z axis followed by 90 degrees around the X axis. This produces the image below.



The object's X-Y plane becomes parallel to the display's surface. The visibility of the diagonal line proves this to be the case. If the X axis rotation occurred around the rotated X axis, the object's X-Y plane would become perpendicular to the display's surface, making the diagonal line invisible.

Of interest is a particular orthographic projection. A rotation around the Z axis of 45 degrees followed by a rotation around the X axis of 35.2 degrees produces an isometric projection, common to a variety of drawing applications.

For reference, a complete listing of the orthographic projection example is given:

```
100 INIT
110 DIM O(11,3),P(3),Q(2,3)
120 RESTORE
130 REM ****************
140 REM  FILL 3D DATA ARRAY
150 O=0
160 DATA 0.5,1,0.5,1,1
170 READ O(1,1),O(1,2),O(2,1),O(4,2),O(5,1)
180 DATA 1,1,1,1
190 READ O(5,2),O(6,1),O(8,3),O(9,1)
200 DATA 1,1,1,1
210 READ O(9,2),O(9,3),O(10,1),O(10,2)
220 REM  FIND MIN AND MAX OF 3D DATA
230 Q=-1.0E+300
240 DATA 1.0E+300,1.0E+300,1.0E+300
250 READ Q(1,1),Q(1,2),Q(1,3)
260 FOR I=1 TO 10
270 FOR J=1 TO 3
280 Q(1,J)=Q(1,J) MIN O(I,J)
290 Q(2,J)=Q(2,J) MAX O(I,J)
300 NEXT J
310 NEXT I
320 REM ****************
330 REM  SPECIFY 3D PARAMETERS - THREE  ROTATION ANGLES
340 DATA 35.2,0,25
350 READ A1,A2,A3
360 REM  INITIALIZE 3D  ROUTINE
370 GOSUB 4000
380 REM ****************
390 REM  FIND MIN AND MAX IN VIEWPLANE'S COORDINATE SYSTEM
400 DATA 1.0E+300,-1.0E+300,1.0E+300,-1.0E+300
410 READ X1,X2,Y1,Y2
420 FOR I=1 TO 2
430 FOR J=1 TO 2
440 FOR K=1 TO 2
450 P(1)=Q(I,1)
460 P(2)=Q(J,2)
470 P(3)=Q(K,3)
480 GOSUB 5000
490 X1=X1 MIN X
500 X2=X2 MAX X
510 Y1=Y1 MIN Y
520 Y2=Y2 MAX Y
530 NEXT K
540 NEXT J
550 NEXT I
560 REM SET UP VIEWPORT AND 2D WINDOW
570 VIEWPORT 0,130,0,100
580 IF (X2-X1)/(Y2-Y1)>1.3 THEN 610
590 WINDOW (X2+X1)*0.5-0.65*(Y2-Y1),(X2+X1)*0.5+0.65*(Y2-Y1),Y1,Y2
600 GO TO 630
610 WINDOW X1,X2,(Y2+Y1)*0.5-(X2-X1)/2.6,(Y2+Y1)*0.5+(X2-X1)/2.6
620 REM ********************
630 REM  MOVE TO FIRST POINT
640 P(1)=O(1,1)
650 P(2)=O(1,2)
660 P(3)=O(1,3)
```

```
670 GOSUB 5000
680 MOVE X,Y
690 REM  DRAW OBJECT
700 FOR I=2 TO 11
710 P(1)=O(I,1)
720 P(2)=O(I,2)
730 P(3)=O(I,3)
740 GOSUB 5000
750 DRAW X,Y
760 NEXT I
770 HOME
780 END

4000 SET DEGREES
4010 C1=COS(A1)
4020 S1=SIN(A1)
4030 C2=COS(A2)
4040 S2=SIN(A2)
4050 C3=COS(A3)
4060 S3=SIN(A3)
4070 RETURN
5000 REM TRANSFORM P(1),P(2),P(3) INTO X,Y
5010 X=C2*C3*P(1)-C2*S3*P(2)+S2*P(3)
5020 Y=(S1*S3-C1*S2*C3)*P(1)+(S1*C3+C1*S2*S3)*P(2)+C1*C2*P(3)
5030 RETURN
```

Listed below are the equations which cause rotation first around the X axis, then around the Y axis, then around the Z axis. These can be substituted for similarly numbered statements in the listing above.

```
5000 REM TRANSFORM P(1),P(2),P(3) INTO X,Y
5010 X=C3*C2*P(1)+(C3*S2*S1-S3*C1)*P(2)+(C3*S2*C1+S3*C1)*P(3)
5020 Y=-S2*P(1)+C2*S1*P(2)+C2*C1*P(3)
5030 RETURN
```

TRUE PERSPECTIVE

The only transformation which does not introduce artifical distortion is a true perspective transformation. The controlling program in the perspective example (statements 100 through 780) is identical to that of the previous orthographic example. The only exceptions are statements 340 and 350, a DATA statement and a READ statement.

```
340 DATA 0.8,-1,0.6,0.5,0.5,0.5
350 READ E(1),E(2),E(3),L(1),L(2),L(3)
```

These are the parameters used to initialize the transformation. The array E contains the location in three dimensional space of the observer's eye. The array L contains the location in three dimensional space of the point toward which the eye is looking. This point should be close to the center of the scene being viewed. This will ensure the minimizing of a certain type of distortion, the kind seen at the edges of photographs taken with extreme wide angle lenses.

In order to depict scenes in true perspective, a Renaissance artist would observe his subject through a pane of glass, marking where the lines between his eye and his subject passed through the pane.

The algorithm in this example duplicates that process mathematically:

```
320 REM ******************
330 REM  SPECIFY 3D PARAMETERS - EYE POSITION & POINT TO BE LOOKED AT
340 DATA 0.8,-1,0.6,0.5,0.5,0.5
350 READ E(1),E(2),E(3),L(1),L(2),L(3)
360 REM  INITIALIZE 3D  ROUTINE
370 GOSUB 4000
380 REM ******************

4000 REM  3D TO 2D PERSPECTIVE INITIALIZATION
4010 REM  MUST BE RUN BEFORE USING TRANSFORM ROUTINE AT 5000
4020 REM    PURPOSE:  1. FIND LOCATION AND ORIENTATION OF VIEWPLANE
4030 REM              2. DEFINE 2D COORDINATE SYSTEM ON VIEWPLANE
4040 REM  E  IS VIEWER'S EYE LOCATION IN 3 SPACE
4050 REM  L  IS 3 SPACE POINT BEING LOOKED AT BY VIEWER'S EYE
4060 DELETE T1,T2,T3,T4,T5
4070 DIM T1(3),T2(3),T3(3),T4(3),T5(3)
4080 IF E(1)<>L(1) OR E(2)<>L(2) THEN 4120
4090 PRINT "E AND L LIE IN SAME VERTICAL LINE "
4100 PRINT "PLEASE CHANGE ONE OF THE FOLLOWING:  E(1),E(2),L(1) OR L(2)"
4110 END
4120 REM
4130 T3=E-L
4140 REM  T3 IS NOW VECTOR FROM L TO E
4150 T7=0
4160 FOR T9=1 TO 3
4170 T7=T7+T3(T9)↑2
4180 NEXT T9
4190 T7=SQR(T7)
4200 REM  T7 IS NOW DISTANCE FROM L TO E
4210 T2=T3/T7
4220 REM  T2 CONTAINS DIRECTION COSINES OF NORMAL VECTOR TO VIEWPLANE
4230 REM
4240 REM  PLACE VIEWPLANE HALF WAY BETWEEN E AND L
4250 T6=T7/2
4260 REM  T6 IS DISTANCE FROM E TO VIEWPLANE


4270 T3=T3*(T6/T7)
4280 T1=E-T3
4290 REM  T1 IS THE INTERSECTION OF VIEWPLANE AND LINE FROM E TO L
4300 REM  DEFINE T1 TO BE ORIGIN OF VIEWPLANE'S 2D COORDINATE SYSTEM
4310 REM
4320 REM  NOW DEFINE "UP" IN VIEWPLANE'S 2D COORDINATE SYSTEM
4330 T3=L
4340 T3(3)=T3(3)+1
4350 REM  T3 IS NOW ENDPOINT OF THE VECTOR (0,0,1) RELATIVE TO L
4360 REM  PROJECT ONTO VIEW PLANE
4370 GOSUB 5170
4380 REM T3 IS NOW 3 SPACE POINT DEFINED TO BE "ABOVE" PLANE'S ORIGIN
4390 REM
4400 T7=0
4410 FOR T9=1 TO 3
4420 T7=T7+(T3(T9)-T1(T9))↑2
4430 NEXT T9
4440 T7=SQR(T7)
4450 REM  T7 IS NOW DISTANCE FROM T1 TO T3
4460 REM
4470 T3=T3-T1
4480 T5=T3/T7
4490 REM  T5 NOW CONTAINS DIRECTION COSINES OF VIEWPLANE'S "VERTICAL"
```

```
4500 REM  CROSS PRODUCT WITH VIEWPLANE'S NORMAL VECTOR IS "HORIZONTAL"
4510 T4(1)=T5(2)*T2(3)-T5(3)*T2(2)
4520 T4(2)=T5(3)*T2(1)-T5(1)*T2(3)
4530 T4(3)=T5(1)*T2(2)-T5(2)*T2(1)
4540 REM  T4 CONTAINS DIRECTION COSINES OF VIEWPLANE'S "HORIZONTAL"
4550 RETURN
4560 REM

5000 REM  ENTRY POINT FOR 3D TO 2D TRANSFORM
5010 REM  P CONTAINS 3 SPACE COORDINATES OF POINT
5020 REM  TO BE TRANSFORMED TO 2D COORDINATES OF VIEWPLANE
5030 T3=P
5040 REM  PROJECT INPUT POINT ONTO VIEWPLANE
5050 GOSUB 5170
5060 REM
5070 REM  COMPUTE 2D COORDINATES OF PROJECTED POINT
5080 X=0
5090 Y=0
5100 FOR T9=1 TO 3
5110 X=X+(T3(T9)-T1(T9))*T4(T9)
5120 Y=Y+(T3(T9)-T1(T9))*T5(T9)
5130 NEXT T9
5140 RETURN
5150 REM  INTERNAL ROUTINE:  FIND 3 SPACE POINT WHERE LINE FROM
5160 REM     T3 TO E INTERSECTS VIEWPLANE
5170 T7=0
5180 T8=0
5190 FOR T9=1 TO 3
5200 T7=T7+(T1(T9)-T3(T9))*T2(T9)
5210 T8=T8+(E(T9)-T3(T9))*T2(T9)
5220 NEXT T9
5230 IF T8>0 THEN 5260
5240 PRINT "E IS TOO CLOSE TO POINT BEING VIEWED"
5250 END
5260 T7=T7/T8
5270 FOR T9=1 TO 3
5280 T3(T9)=(E(T9)-T3(T9))*T7+T3(T9)
5290 NEXT T9
5300 RETURN
```

```
5040 REM  PROJECT INPUT POINT ONTO VIEWPLANE
5050 GOSUB 5170
5060 REM
5070 REM  COMPUTE 2D COORDINATES OF PROJECTED POINT
5080 X=0
5090 Y=0
5100 FOR T9=1 TO 3
5110 X=X+(T3(T9)-T1(T9))*T4(T9)
5120 Y=Y+(T3(T9)-T1(T9))*T5(T9)
5130 NEXT T9
5140 RETURN
5150 REM  INTERNAL ROUTINE:  FIND 3 SPACE POINT WHERE LINE FROM
5160 REM     T3 TO E INTERSECTS VIEWPLANE
5170 T7=0
5180 T8=0
5190 FOR T9=1 TO 3
5200 T7=T7+(T1(T9)-T3(T9))*T2(T9)
5210 T8=T8+(E(T9)-T3(T9))*T2(T9)
5220 NEXT T9
5230 IF T8>0 THEN 5260
5240 PRINT "E IS TOO CLOSE TO POINT BEING VIEWED"
5250 END
5260 T7=T7/T8
5270 FOR T9=1 TO 3
5280 T3(T9)=(E(T9)-T3(T9))*T7+T3(T9)
5290 NEXT T9
5300 RETURN
```

The initialization section (statements 4000 through 4550) has one error termination: statement 4110. The routine must have a direction to call "up". In statement 4340, a line pointing up is defined to be parallel to the Z axis, with "up" being the positive Z direction. If the line between E and L is parallel to the Z axis, the program cannot define which way is up. The IF statement at line 4080 prevents this from happening.

The transformation section also has one error termination: statement 5250. If a point in the object being viewed is at or behind E, the transformation produces inappropriate results. The check at statement 5230 prevents this occurrence.

Both the initialization and transformation sections include REMARK statements which sufficiently describe the mathematical operations being performed. A background in the appropriate mathematics will permit a complete understanding or the algorithm used in these routines.

A complete listing of the perspective projection program follows. To help in understanding the algorithm used, an excessive number of REMARK statements have been included. These are superfluous and may be removed without affecting program execution.

```
100 INIT
110 DIM E(3),L(3),O(11,3),P(3),Q(2,3)
120 RESTORE
130 REM ******************
140 REM  FILL 3D DATA ARRAY
150 O=0
160 DATA 0.5,1,0.5,1,1
170 READ O(1,1),O(1,2),O(2,1),O(4,2),O(5,1)
180 DATA 1,1,1,1
190 READ O(5,2),O(6,1),O(8,3),O(9,1)
200 DATA 1,1,1,1
210 READ O(9,2),O(9,3),O(10,1),O(10,2)
220 REM  FIND MIN AND MAX OF 3D DATA
230 Q=-1.0E+300
240 DATA 1.0E+300,1.0E+300,1.0E+300
250 READ Q(1,1),Q(1,2),Q(1,3)
260 FOR I=1 TO 10
270 FOR J=1 TO 3
280 Q(1,J)=Q(1,J) MIN O(I,J)
290 Q(2,J)=Q(2,J) MAX O(I,J)
300 NEXT J
310 NEXT I
320 REM ******************
330 REM  SPECIFY 3D PARAMETERS - EYE POSITION & POINT TO BE LOOKED AT
340 DATA 0.8,-1,0.6,0.5,0.5,0.5
350 READ E(1),E(2),E(3),L(1),L(2),L(3)
360 REM  INITIALIZE 3D  ROUTINE
370 GOSUB 4000
380 REM ******************
390 REM  FIND MIN AND MAX IN VIEWPLANE'S COORDINATE SYSTEM
400 DATA 1.0E+300,-1.0E+300,1.0E+300,-1.0E+300
410 READ X1,X2,Y1,Y2
420 FOR I=1 TO 2
430 FOR J=1 TO 2
440 FOR K=1 TO 2
450 P(1)=Q(I,1)
460 P(2)=Q(J,2)
470 P(3)=Q(K,3)
480 GOSUB 5000
490 X1=X1 MIN X
500 X2=X2 MAX X
510 Y1=Y1 MIN Y
520 Y2=Y2 MAX Y
530 NEXT K
540 NEXT J
550 NEXT I
560 REM SET UP VIEWPORT AND 2D WINDOW
570 VIEWPORT 0,130,0,100
580 IF (X2-X1)/(Y2-Y1)>1.3 THEN 610
590 WINDOW (X2+X1)*0.5-0.65*(Y2-Y1),(X2+X1)*0.5+0.65*(Y2-Y1),Y1,Y2
600 GO TO 630
610 WINDOW X1,X2,(Y2+Y1)*0.5-(X2-X1)/2.6,(Y2+Y1)*0.5+(X2-X1)/2.6
620 REM ********************
630 REM  MOVE TO FIRST POINT
640 P(1)=O(1,1)
650 P(2)=O(1,2)
660 P(3)=O(1,3)
670 GOSUB 5000
680 MOVE X,Y
690 REM  DRAW OBJECT
```

```
700 FOR I=2 TO 11
710 P(1)=O(I,1)
720 P(2)=O(I,2)
730 P(3)=O(I,3)
740 GOSUB 5000
750 DRAW X,Y
760 NEXT I
770 HOME
780 END
```

```
4000 REM  3D TO 2D PERSPECTIVE INITIALIZATION
4010 REM  MUST BE RUN BEFORE USING TRANSFORM ROUTINE AT 5000
4020 REM    PURPOSE:  1. FIND LOCATION AND ORIENTATION OF VIEWPLANE
4030 REM              2. DEFINE 2D COORDINATE SYSTEM ON VIEWPLANE
4040 REM  E  IS VIEWER'S EYE LOCATION IN 3 SPACE
4050 REM  L  IS 3 SPACE POINT BEING LOOKED AT BY VIEWER'S EYE
4060 DELETE T1,T2,T3,T4,T5
4070 DIM T1(3),T2(3),T3(3),T4(3),T5(3)
4080 IF E(1)<>L(1) OR E(2)<>L(2) THEN 4120
4090 PRINT "E AND L LIE IN SAME VERTICAL LINE "
4100 PRINT "PLEASE CHANGE ONE OF THE FOLLOWING:  E(1),E(2),L(1) OR L(2)"
4110 END
4120 REM
4130 T3=E-L
4140 REM  T3 IS NOW VECTOR FROM L TO E
4150 T7=0
4160 FOR T9=1 TO 3
4170 T7=T7+T3(T9)↑2
4180 NEXT T9
4190 T7=SQR(T7)
4200 REM  T7 IS NOW DISTANCE FROM L TO E
4210 T2=T3/T7
4220 REM  T2 CONTAINS DIRECTION COSINES OF NORMAL VECTOR TO VIEWPLANE
4230 REM
4240 REM  PLACE VIEWPLANE HALF WAY BETWEEN E AND L
4250 T6=T7/2
4260 REM  T6 IS DISTANCE FROM E TO VIEWPLANE
4270 T3=T3*(T6/T7)
4280 T1=E-T3
4290 REM  T1 IS THE INTERSECTION OF VIEWPLANE AND LINE FROM E TO L
4300 REM  DEFINE T1 TO BE ORIGIN OF VIEWPLANE'S 2D COORDINATE SYSTEM
4310 REM
4320 REM  NOW DEFINE "UP" IN VIEWPLANE'S 2D COORDINATE SYSTEM
4330 T3=L
4340 T3(3)=T3(3)+1
```

```
4350 REM  T3 IS NOW ENDPOINT OF THE VECTOR (0,0,1) RELATIVE TO L
4360 REM  PROJECT ONTO VIEW PLANE
4370 GOSUB 5170
4380 REM T3 IS NOW 3 SPACE POINT DEFINED TO BE "ABOVE" PLANE'S ORIGIN
4390 REM
4400 T7=0
4410 FOR T9=1 TO 3
4420 T7=T7+(T3(T9)-T1(T9))↑2
4430 NEXT T9
4440 T7=SQR(T7)
4450 REM  T7 IS NOW DISTANCE FROM T1 TO T3
4460 REM
4470 T3=T3-T1
4480 T5=T3/T7
4490 REM  T5 NOW CONTAINS DIRECTION COSINES OF VIEWPLANE'S "VERTICAL"
4500 REM  CROSS PRODUCT WITH VIEWPLANE'S NORMAL VECTOR IS "HORIZONTAL"
4510 T4(1)=T5(2)*T2(3)-T5(3)*T2(2)
4520 T4(2)=T5(3)*T2(1)-T5(1)*T2(3)
4530 T4(3)=T5(1)*T2(2)-T5(2)*T2(1)
4540 REM  T4 CONTAINS DIRECTION COSINES OF VIEWPLANE'S "HORIZONTAL"
4550 RETURN
4560 REM
5000 REM  ENTRY POINT FOR 3D TO 2D TRANSFORM
5010 REM  P CONTAINS 3 SPACE COORDINATES OF POINT
5020 REM  TO BE TRANSFORMED TO 2D COORDINATES OF VIEWPLANE
5030 T3=P
5040 REM  PROJECT INPUT POINT ONTO VIEWPLANE
5050 GOSUB 5170
5060 REM
5070 REM  COMPUTE 2D COORDINATES OF PROJECTED POINT
5080 X=0
5090 Y=0
5100 FOR T9=1 TO 3
5110 X=X+(T3(T9)-T1(T9))*T4(T9)
5120 Y=Y+(T3(T9)-T1(T9))*T5(T9)

5130 NEXT T9
5140 RETURN
5150 REM  INTERNAL ROUTINE:  FIND 3 SPACE POINT WHERE LINE FROM
5160 REM     T3 TO E INTERSECTS VIEWPLANE
5170 T7=0
5180 T8=0
5190 FOR T9=1 TO 3
5200 T7=T7+(T1(T9)-T3(T9))*T2(T9)
5210 T8=T8+(E(T9)-T3(T9))*T2(T9)
5220 NEXT T9
5230 IF T8>0 THEN 5260
5240 PRINT "E IS TOO CLOSE TO POINT BEING VIEWED"
5250 END
5260 T7=T7/T8
5270 FOR T9=1 TO 3
5280 T3(T9)=(E(T9)-T3(T9))*T7+T3(T9)
5290 NEXT T9
5300 RETURN
```

Appendix A

REFERENCE MATERIAL

GLOSSARY

TERM	DEFINITION
Accumulator	A temporary storage area used for storing a number, summing it with another number, and replacing the first number with the sum.
Algorithm	A step-by-step method for solving a given problem.
Argument	A value operated on by a function or a keyword. Also called a parameter.
Arithmetic Operator	Operators which describe arithmetic operations, such as +, −, /, ↑.
Array	A collection of data items arranged in a meaningful pattern. In the Graphic System, arrays may be one or two dimensional; that is, organized into rows, or rows and columns.
Array Variable	A name corresponding to a (usually) multi-element collection of data items. Array variables may be named with the characters A through Z and A0 through Z9.
ASCII Code	A standardized code of alphanumeric characters, symbols, and special "control" characters. ASCII is an acronym for American Standard Code for Information Interchange.
Assignment Statement	A statement which is used to assign, or give, a value to a variable.
BASIC	An acronym derived from Beginners All-purpose Symbolic Instruction Code. BASIC is a "high level" programming language because it uses English-like instructions.
Binary String	A connected sequence of 1's and 0's.
Bit	A Binary digit. A unit of data in the binary numbering system; a 1 or 0.
Byte	A group of consecutive binary digits operated upon as a unit. One ASCII character, for example, is represented by one binary byte.
Character String	A connected sequence of ASCII characters, sometimes referred to as simply "string".

TERM	DEFINITION
Coding	The process of preparing a list of successive computer instructions for solving a specific problem. Coding is usually done from a flowchart or algorithm.
Concatenate	To join together two character strings with the concatenation operator (&) forming a larger character string.
Constant	A number that appears in its actual numerical form. In the following expression, 4 is a constant: X = 4 * P
CRT	An abbreviation for Cathode Ray Tube. In the Graphic System, the CRT is a "storage" display, as opposed to a "refreshed" or TV-like display.
Cursor	The flashing rectangular dot matrix on the Graphic System display that is located at the position of the "next" character to be printed.
Debug	The process of locating and correcting errors in a program; also, the process of testing a program to ensure that it operates properly.
Default	The property of a computer that enables it to examine a statement requiring parameters, to see if those parameters are present; and, finding none, assigning substitute values for those parameters. Default actions provide a powerful means for saving memory space and time when program statements are entered into memory.
Delimiter	A character that fixes the limits, or bounds, of a string of characters.
Dyadic	Refers to an operator having two operands.
Execute	To perform the operations indicated by a statement or group of statements.
Expression	Refers to either numeric expressions or string expressions. A collection of variables, constants, and functions connected by operators in such a way that the expression as a whole can be reduced to a constant.
Flowchart	A programming tool that provides a graphic representation of a routine to solve a specific problem.
Function	A special purpose operation referring to a set of calculations within an expression, as in the sine function, square root function, etc.

TERM	DEFINITION
Graphic Display Unit (GDU)	An internal unit of measure representing one one-hundredth of the vertical axis on the graphic drawing surface.
Graphics	Computer output that is composed of lines rather than letters numbers, and symbols.
Hardware	The physical devices and components of a computer.
Index	A number used to identify the position of a specific quantity in an array or string of quantities. That is, in the array A, the elements are represented by the variables A(1), A(2), . . . A(50); the indexes are 1, 2, . . . 50.
Input	Data that is transferred to the Graphic System memory from an external source.
Instruction	A line number plus a statement (i.e., A line number plus a keyword plus any associated parameters).
Integer	A whole number; a number without a decimal part.
Interrupt	To cause an operation to be halted in such a fashion that it can be resumed at a later time.
Iterate	To repeatedly execute a series of instructions in a loop until a condition is satisfied.
Justify	To align a set of characters to the right or left of a reference point.
Keyboard	The device that encodes data when keys are pressed.
Keyword	An alphanumeric code that the Graphics System recognizes as a function to be performed.
Line Number	An integer establishing the sequence of execution of lines in a program. In the Graphic System, line numbers must be in the range of 1 through 65,535.
Logic	In the Graphic System, the principle of truth tables, also, the interconnection of on-off, true-false elements, etc., for computational purposes.
Logical Expression	A numeric expression using the logical operators AND, OR, and NOT. The numeric expression is arranged in such a way that the numeric result is a logical 1 or a logical 0. A logical expression may be part of a larger numeric expression involving relational operators and/or arithmetic operators.
Logical Operator	Operators which return logical 1's and 0's, specifically, the AND, OR, and NOT operators. "True" operations return "1", "false" operations return "0".

TERM	DEFINITION
Loop	Repeatedly executing a series of statements for a specified number of times. Also, a programming technique that causes a group of statements to be repeatedly executed.
Mantissa	In scientific notation, the term mantissa refers to that part of the number which precedes the exponent. For example, the mantissa in the number 1.234E+200 is 1.234.
Matrix	A rectangular array of numbers subject to special mathematical operations. Also, something having a rectangular arrangement of rows and columns.
Memory	This generally refers to the Read/Write Random Access Memory that contains BASIC programs and data, as opposed to the Read Only Memory which contains the BASIC interpreter.
Monadic	Refers to an operator that has only one operand.
Numeric Constant	Any real number that is entered as numeric data; also, the contents of a numeric variable.
Numeric Expression	Any combination of numeric constants, numeric variables, array variables, subscripted array variables, numeric functions, or string relational comparisons inclosed in parentheses, joined together by one or more arithmetic, logical, or relational operators in such a way that the expression, as a whole, can be reduced to a single numeric constant when evaluated.
Numeric Function	Special purpose mathematical operations which reduce their associated parameters (or arguments) to a numeric constant.
Numeric Variable	A variable that can contain a single numeric value. Numeric variables can be named with the characters A through Z and A0 through Z9, and can be used in numeric expressions.
Operand	Any one of the quantities involved in an operation. Operands may be numeric expressions or constants. In the numeric expression A = B+4*C, the numeric variables B and C, and the numeric constant 4 are operands.
Operator	A symbol indicating the operation to be performed on two operands. That is, in the expression Z + Y, the plus sign (+) is the operator.
Output	The results obtained from the Graphic System; also, information transferred to a peripheral device.

TERM	DEFINITIONS
Parameter	A quantity that may be specified as different values; usually used in conjunction with BASIC statements. For example, in the statement WINDOW −50, 50−100, 100, the parameters are −50, 50, −100, and 100.
Peripheral Device	Various devices (Hard Copy Unit, Plotter, Magnetic Tape Drive, etc.) that are used in the Graphic System to input data, output data, and store data.
Program	A sequence of instructions for the automatic solution of a problem, resulting from a planned approach.
Programming	The process of preparing programs from the standpoint of first planning the process from input to output, and then entering the code into memory.
Relational Operator	An operator that causes a comparison of two operands and returns a logical result. Comparisons that are "true" return a "1", comparisons that are "false" return a "0". The relational operators in the Graphic System are =, <>, <, >, =>, and <=.
ROM	Read Only Memory. The ROM is that portion of the system memory that can not be changed. The information in the ROM can only be read. In the Graphic System, the BASIC operating system resides in a ROM.
Scientific Notation	A format representing numbers as a fractional part, or mantissa, and a power of 10, or characteristic, as in 1.23E45.
Software	Prepared programs that simplify computer operations, such as mathemetics and statistics software. Software must be reloaded into memory each time the system power is turned on.
Statement	A keyword plus any associated parameters.
String	A connected sequence of alphanumeric characters. Often called a character string.
String Constant	A string of characters of fixed length enclosed in quotation marks; also, the contents of a string variable.
String Function	Special purpose functions that manipulate character strings and produce string constants.
String Variable	A variable that contains only alphanumeric characters, or "strings". String variables can be represented by the symbols A$ through Z$. They have a default length of 72; i.e., they can contain up to 72 characters without being dimensioned in a DIM statement.

TERM	DEFINITION
Subroutine	A part of a larger "main" routine, arranged in such a way that control is passed from the main routine to the subroutine. At the conclusion of the subroutine, control returns to the main routine. Control is usually passed to the subroutine from more than one place in the main routine.
Subscripted Array Variable	An array variable followed by one or two subscripts, as in A(9), B3(1,2), and Z(N). The subscripts refer to a specific element within the array.
Substring	A portion of a larger string; "BC", for example, is a substring within the string "ABCD".
System	A purposeful collection of interacting components (hardware and software) forming an organized whole and performing a function beyond the capability of any one component.
Truncate	To reduce the number of least significant digits present in a number, in contrast to rounding off. For example, the number 5 is the result of truncating the decimal part of the number 5.382.
User Data Units	The units of measure the programmer elects to work with for a particular graphing application. These units are established in the WINDOW statement as a numeric range for each axis. For example the vertical axis range can be set starting at 0 "dollars" and ending at 100 "dollars;" the horizontal range can be set starting at the "year" 1962 and ending at the year "1975." All coordinate values for graphic statements are specified in user data units (except VIEWPORT).
Variable	A symbol, corresponding to a location in memory, whose value may change as a program executes.
Variable Name	A name selected by the programmer that represents a specific variable. Numeric variables and array variables may be named with the characters A through Z and A0 through Z9. String variables may be named with the characters A$ through Z$.

ERROR MESSAGES

MESSAGE NUMBER	ERROR MESSAGE
Ø	A system error Ø is generated whenever a firmware failure condition occurs. An INIT command and a DELETE ALL command are issued to recover firmware program control. An example of firmware failure condition occurs when a program is stored on magnetic tape in binary format from a 4052 Graphic System, and the program contains commands not available on the 4051 Graphic System. When such a program is loaded into the 4051 Graphic System, the firmware fails, and a system error Ø results.
1	An arithmetic operation has resulted in an out of range number. Example: 1/1.0E-308
2	A divide by zero operation has resulted in an out of range number. Example: 4/0
3	An exponentiation operation has resulted in an out of range number. Example: 5↑1.0E+300
4	An exponentiation operation involving the base e has resulted in an out of range number. Example: EXP (1.0E+234)
5	The parameter of a trigonometric function is too large. That is, the variable N in the statement A=SIN(N*2*PI) is greater than 65536. Example: A=SIN(4.2E+5) when the trigonometric units are set to RADIANS.
6	An attempt has been made to take the square root of a negative number. The positive square root is returned by default. Example: SQR (–4)
7	The line number in the program line is not an integer within the range 1 to 65535. Example: Ø REM THIS IS AN INVALID LINE NUMBER

MESSAGE NUMBER	ERROR MESSAGE
8	The matrix arrays are not conformable in the current math operation. That is, they are not of the same dimension and/or do not have the same number of elements. Example: INIT DIM A(2),B(2),C(3) A=1 B=2 C=A+B
9	A previously defined numeric variable can not be dimensioned as an array variable without deleting the numeric variable first. Example: INIT B=3 DIM B(2,2)
10	There is an error in the subscript of a variable due to one of the following reasons: 1. A numeric variable can't be subscripted. 2. A subscript is out of range. Example 1: INIT B=3 PRINT B(4) Example 2: INIT DIM A(2,2) A(2,3)=5
11	An attempt has been made to use an undefined DEF FN function.
12	There is a parameter error in the CALL statement to a ROM pack.
13	A WBYTE parameter is not within the range –255 through +255. Example: WBYTE 300
14	A parameter for the APPEND statement is invalid.
15	An attempt has been made to APPEND to a non-existent line number.
16	There is an invalid parameter in the FUZZ statement. Example: FUZZ 0

MESSAGE NUMBER	ERROR MESSAGE
17	There is an invalid parameter in a RENUMBER operation due to one of the following reasons: 1. The first or third parameter is not a line number within the range 1 through 65535. 2. The increment (second parameter) is not within the range 1 through 65535 or is so large that out of range line numbers are generated during the RENUMBER operation. 3. Statement replacement or statement interlacing will occur if the RENUMBER operation is attempted. This error may occur during an APPEND operation.
18	Not used.
19	There is an invalid parameter in a GOTO, FOR, or NEXT statement. Example: 500 FOR I=1 to 20 where I has been previously defined as an array variable.
20	The logical unit number specified in the statement is not within the range 0 through 9. Example: 100 ON EOF (10) THEN 500
21	The assignment statement is invalid because of one of the following reasons: 1. An attempt has been made to assign an array to a numeric variable. 2. Two arrays in the statement are not conformable (not of the same dimension and/or do not have the same number of elements). 3. An attempt has been made to assign a character string to a string variable and the character string is larger than the dimensioned size of the variable.
22	There is an error in an exponentiation operation because the base is less than 0 and the exponent is not an integer less than 256 Example: −10↑257.5
23	An attempt has been made to take the LOG or LGT of a number which is equal to or less than 0. Example: LOG (−1)

MESSAGE NUMBER	ERROR MESSAGE
24	The parameter of the ASN function or the ACS function is not within the range −1 to +1. Example: ASN (2)
25	The parameter of the CHR function is not within the range 0 through 127. Example: A$=CHR(128)
26	Not used.
27	The parameter is out of the domain of the function. Example: A$=STR(X) where X has been previously defined as an array variable.
28	A REP function parameter is invalid.
29	The parameter in the VAL function is not a character string containing a valid number. Example: A=VAL("Hi")
30	The matrix multiply operation failed because the arrays are not conformable.
31	The matrix inversion failed because the determinant was 0. This error is treated as a SIZE error.
32	The routine name specified in the CALL statement can not be found. Example: CALL "FIX IT" where the routine "FIX IT" resides in a ROM pack which is not plugged into the system.
33	Not used.
34	The DATA statement is invalid because of one of the following reasons: 1. There isn't a DATA statement in the current BASIC program. 2. There is not enough data in the DATA statement from the present position of the pointer to the end of the statement. 3. An attempt has been made to RESTORE the data statement pointer to a nonexistent DATA statement.
35	The statements DEF FN, FOR, and ON. . .THEN. . . can not be entered without a line number.

MESSAGE NUMBER	ERROR MESSAGE
36	There is an undefined variable in the specified line. A numeric variable has not been assigned a value or an array element has not been assigned a value. Example: INIT DIM A(2,2) A(1,2) = 4 PRINT A
37	An extended function ROM (Read Only Memory) is required to perform this operation.
38	This output operation cannot be executed because the current BASIC program is marked SECRET.
39	This operation can not be executed because the Random Access Memory is full. Some program lines or variables must be deleted.
40	Not used.
41	A SIZE interrupt condition has occurred and an ON SIZE THEN statement has not been executed in the current BASIC program.
42	A PAGE FULL interrupt condition has occurred.
43	A peripheral device on the General Purpose Interface Bus is requesting service and an ON SRQ THEN statement has not been executed in the current BASIC program.
44	The EOI signal line on the General Purpose Interface Bus has been activated and an ON EOI THEN statement has not been activated in the current BASIC program.
45	A ROM pack is requesting service and the ON UNIT for external interrupt number 1 has not been activated in the current BASIC program.
46	A ROM pack is requesting service and the ON UNIT for external interrupt number 2 has not been activated in the current BASIC program.
47	A ROM pack is requesting service and the ON UNIT for external interrupt number 3 has not been activated in the current BASIC program.
48	The end of the current file has been reached on an I/O device and an ON EOF THEN statement has not been executed in the current BASIC program.
49	The statement with the specified line number is too long. This error situation occurs if an attempt is made to LIST or SAVE a BASIC program which contains a line with more than 72 characters. Sometimes a RENUMBER operation can make a line longer than 72 characters.

MESSAGE NUMBER	ERROR MESSAGE
50	The incoming BASIC program contains a line with more than 72 characters.
51	The line number specified in this statement cannot be found or is invalid. Example: GO TO 500 where the line 500 doesn't exist or PRINT USING 100: where line 100 isn't an IMAGE statement.
52	Either the specified magnetic tape file doesn't exist or an attempt has just been made to KILL the LAST (dummy) file.
53	After 10 attempts, the internal magnetic tape unit has been unable to read a portion of the current magnetic tape. The tape head has been positioned after the bad portion in the file to allow the rest of the file to be read.
54	The end of the magnetic tape medium has been detected. Marking a file longer than the remaining portion of the tape can cause this error.
55	An attempt has been made to incorrectly access a magnetic tape file. Example: Executing an OLD statement when the tape head is positioned in the middle of a data file.
56	An attempt has been made to send information to a write-protected tape. Remove the tape cartridge, rotate the lock-out plug until the black arrow points away from SAFE, insert the tape cartridge, and try the operation again.
57	An attempt has been made to read to or write to a non-existent tape cartridge. Insert a tape cartridge into the tape slot and try the operation again.
58	An attempt has been made to read data which is stored in an invalid magnetic tape format. The tape format must be compatible with the Graphic System standard.
59	A program was not found when the OLD statement was executed.
60	Not used.
61	An attempt has been made to execute an invalid operation on an open magnetic tape file. Example: Executing a MARK statement with the tape head positioned in the middle of an open data file.
62	There is a Disk File system parameter error.

MESSAGE NUMBER	ERROR MESSAGE
63	There is an error in a binary data header, most likely caused by a machine malfunction.
64	The character string is too long to output in binary format. The length is limited to 8192 characters.
65	A parity error has occurred in the 4052 or 4054 RAM memory. Although the error is nonfatal (and the message will not be repeated), further operations are unreliable until power has been turned off and back on. In the 4051 this error is not used.
66	The primary address in the specified line is not within the range 0 through 255.
67	An attempt has been made to execute an illegal I/O operation on an internal peripheral device. Example: DRAW @33:50,50
68	The diagnostic loader failed.
69	An input error or an output error has occured on the General Purpose Interface Bus. Both the NDAC and NRFD signal lines are inactive high, which is an illegal GPIB state. This usually means that there are no peripheral devices connected to the GPIB.
70	There is an incomplete literal string specification in the format string. Example: 100 IMAGE 6D,5("MARK
71	A format string is not specified for the PRINT USING operation. Example: 100 IMAGE 6D 110 PRINT USING 100: 23,24,25 Line 100 should be: 100 IMAGE 3(6D)
73	There is an invalid character in the format string specified in the PRINT USING statement.
74	An n modifier in the format string is out of range or is incorrectly used. n modifiers must be positive integers within the range 1 through 11 when used with E field operator and must be within the range 1 through 255 when used with A,D,L,P,T,X,",(, and / field operators.

MESSAGE NUMBER	ERROR MESSAGE
75	The format string specified in the PRINT USING statement is too long (i.e., there are too many data specifiers for the PRINT statement). Example: 100 IMAGE 3(6D) 110 PRINT USING 100:A,B Line 100 should be: 100 IMAGE 2(6D)
76	Parentheses are incorrectly used in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 2(6D 110 PRINT USING 100:A,B Line 100 should be 100 IMAGE 2(6D)
77	There is an invalid modifier to a field operator in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 2(6D),2S 110 PRINT USING 100:A,B Line 100 should be: 100 IMAGE 2(6D),S An n modifier is not allowed
78	An S modifier is incorrectly positioned in the format string which is specified in the PRINT USING statement. The S modifier must always be positioned at the end of the format string. Example: 100 IMAGE 4D,S,8A Line 100 should be: 100 IMAGE 4D,8A,S
79	A comma is incorrectly used in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 6,D,S Line 100 should be: 100 IMAGE 6D,S
80	A decimal point is incorrectly used in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE .3D 110 PRINT USING 100:812.345 Line 100 should be: 100 IMAGE FD.3D
81	A data type mismatch has occurred in the PRINT USING statement. Example: 100 IMAGE 6D,6A 110 PRINT USING 100: "MARY",26 Line 100 should be: 100 IMAGE 6A,6D

MESSAGE NUMBER	ERROR MESSAGE
82	A tabbing error has occurred in the format string which is specified in the PRINT USING statement. Example: 100 IMAGE 10A,2T,FD 110 PRINT USING 100: "ENTER DATA",D The absolute tab to position 2 specified by 2T in line 100 cannot occur because the cursor has already advanced beyond position 2. The tab specification must be at least 11T in this case.
83	A number specified in the PRINT USING statement contains an exponent outside the range ± 127. Example: 100 IMAGE FD.3D 110 PRINT USING 100:8.5E+200
84	The IMAGE format string was deleted during the PAGE FULL interrupt routine.
85	A portion of the IMAGE format string was deleted or altered during the PAGE FULL interrupt routine.
86	A portion of the data specified in the PRINT statement was deleted during the PAGE FULL interrupt routine.
87	A data item specified in the PRINT USING statement is too large to fit into the print field specified in the format string. Example: 100 IMAGE 5A 110 PRINT USING 100: "HORSE FEATHERS" In this example, the string constant "HORSE FEATHERS" is too large to fit into the 5 character field which is specified in line 100.
88	Not used.
89	A ROM pack has issued an error message.
90	Not used.
91	Not used.
92	Not used.
93	Not used.
94	Not used.
95	An internal conversion error has occurred because a parameter in the specified statement is negative.
96	An internal conversion error has occurred because a parameter in the specified statement is greater than 65535.

GRAPHICS AS I/O

The Graphic System can be considered to be a "central processing unit" which communicates to various peripheral devices, some of which are internal and some of which are external. Internal peripheral devices are the keyboard, the tape, the display, and the General Purpose Interface Bus. External peripherals are those connected to the Graphic System via the GPIB. The Graphic System is very consistent when ASCII information is sent to or received from any peripheral. Implicit in each Graphic System output command (such as PRINT) are three functions:

Any conversion to be performed on the stored data prior to its output.

Specifying which peripheral is to receive the data.

Specifying the function to be performed by the peripheral.

The PRINT command is an example.

Any data sent to a peripheral via the PRINT command must be converted from internal storage form to ASCII characters. The command itself can be used in three forms:

PRINT

PRINT @P:

PRINT @P,S:

The P in the above forms is the primary address. This number determines which peripheral device will receive the data.

When no primary address is specified (as in the first form of PRINT, above), the PRINT command itself specifies a default primary address of 32, signifying the display. When no primary address is specified, PRINT sends data to the display.

The S in the above form is the secondary address. This tells the peripheral what type of data to expect and what to do with it. The default secondary address for PRINT is 12, which tells the display that it will receive ASCII characters and that these characters are to be printed on the display in their ASCII form.

Another ASCII output command is DRAW, which performs the same three types of functions as PRINT:

Convert data from internal to external form.

Determine device if no primary address is specified.

Determine device function if no secondary address is specified.

Graphic information is sent to any graphic device as GDU's expressed in the form of ASCII characters. An important point is that the DRAW command outputs graphic information in the same fundamental form that the PRINT command outputs ASCII characters. It is useful to compare the DRAW command with the PRINT command. They are listed below with their default primary and secondary addresses:

```
PRINT @32,12;          DRAW @32,20:
```

Both commands output ASCII characters. They both send this ASCII information to the display (primary address 32). There are, however, two fundamental differences: what is done to the information before it is sent out (determined by the keyword) and what the display does with the information when received (determined by the secondary address).

The secondary address 12 tells the display to print the characters which are equivalent to the ASCII characters it receives. The secondary address 20 tells the display to draw a line from wherever the graphic point is currently located to a location on the display specified (in GDU's) by the ASCII characters received.

Enter the following into the Graphic System:

```
PRINT @32,12: 65,50
```

This command instructs the Graphic System to output the ASCII characters 6 5 and 5 0. The primary address of 32 means that characters are sent to the display. The secondary address of 12 instructs the display to print the characters as received. Now enter the following command:

```
PRINT @32, 20: 65,50
```

This command causes a line to be drawn to the center of the display. The only difference between the two commands is the secondary address. Both commands send the same ASCII characters to the display. However, the same characters are interpreted differently because of the different secondary address.

Now enter the following commands into the Graphic System:

```
INIT
WINDOW –50,50,–50,50
DRAW @32,20: 0,0
```

The DRAW command causes a line to be drawn to the origin of the user data space defined by the WINDOW command. The INIT and WINDOW commands (above) defined the user data space such that the center of the display corresponds to the location (0,0) in that data space. Therefore, a DRAW 0,0 in this context really is saying "draw a line to the center of the display". The center of the display is the location 65,50 in GDU's. So the DRAW command must convert user data units into GDU's prior to outputting the graphic information. A secondary address of 20 specifies that the ASCII information received is to be interpreted by the display as graphic information.

If a DRAW command with a secondary address which is not 20 is executed, the display will interpret the information it receives differently.

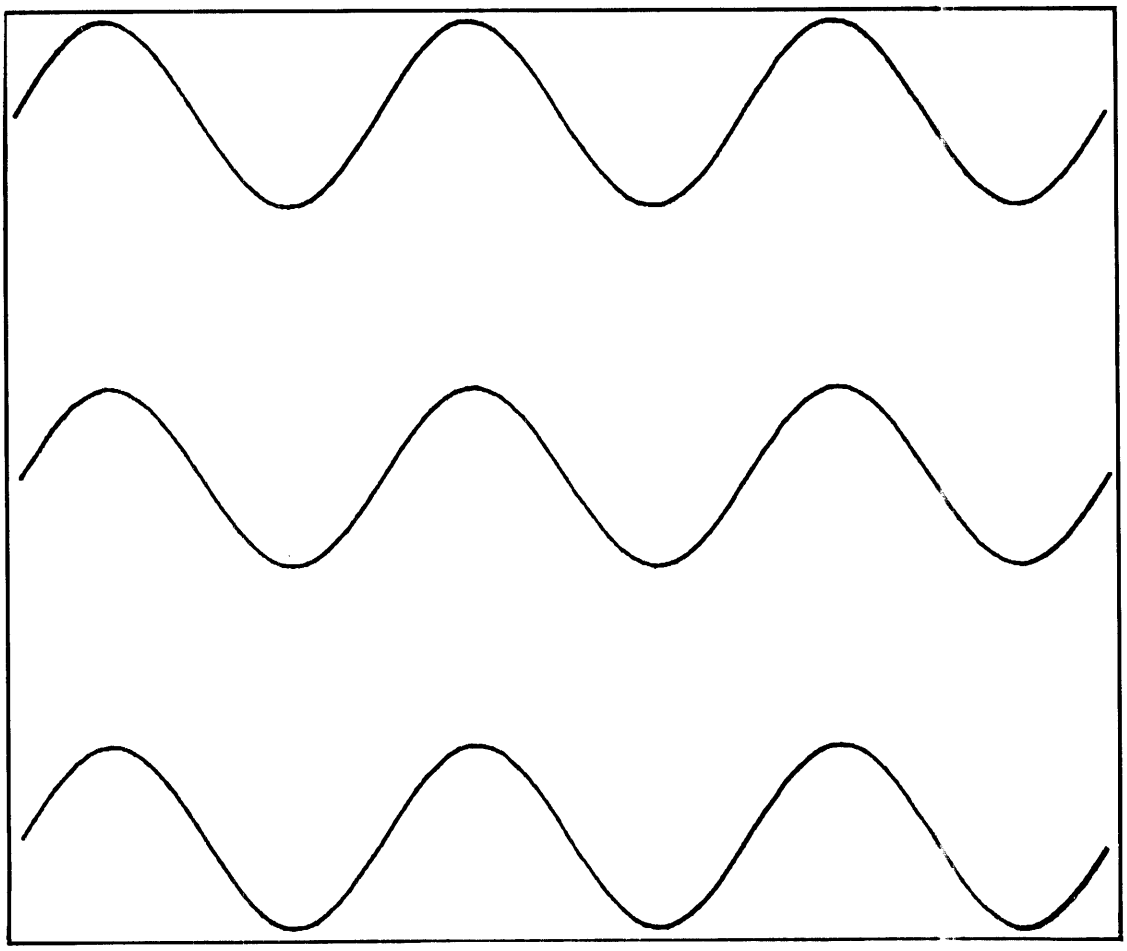
Now enter the following command into the Graphic System:

```
DRAW @32,12: 0,0
```

As shown above, a secondary address of 12 instructs the display to interpret the information it receives only as ASCII characters, not as graphic information. When the command DRAW @32,12: 0,0 is executed, the numbers 65 and 50 are printed on the display. This shows that the user data space location of (0,0) has been transformed into the display location of (65,50).

The above commands demonstrate two principles. One: that graphics are really device independent mathematics until the information is sent to a graphic device. Two: graphic commands perform several operations on graphic data before it is output. The example below shows that these operations require varying amounts of time to perform. (When run, the program below calculates for several seconds before any output is displayed. After the program has been run once, pressing user definable key one causes the output to be repeated without re-performing the calculations.) Enter the following statements into the Graphic System:

```
1 GO TO 100
4 GO TO 310
100 N=100
110 D=1080/N
120 DELETE A,B,X,Y,Z
130 DIM A(N),B(N),X(N),Y(N),Z(N,2)
140 SET DEGREES
150 FOR I=1 TO N
160 X(I)=I*D
170 Y(I)=SIN(I*D)+1
180 NEXT I
190 A=D
200 B(1)=Y(1)-1
210 FOR I=2 TO N
220 B(I)=Y(I)-Y(I-1)
230 NEXT I
240 D1=130/N
250 FOR I=1 TO N
260 Z(I,1)=I*D1
270 Z(I,2)=Y(I)*10
280 NEXT I
290 PAGE
300 PRINT "G"
310 VIEWPORT 0,130,80,100
320 WINDOW 0,1080,0,2
330 MOVE 0,1
340 RDRAW A,B
350 VIEWPORT 0,130,40,60
360 MOVE 0,1
370 DRAW X,Y
380 PRINT @32,21:0,10
390 PRINT @32,20:Z
400 END
```



The above program demonstrates that there are three fundamental ways to draw a line on the Graphic System display: RDRAW (statement 340), DRAW (statement 370) and PRINT (statement 390). The program also shows that each of these three commands requires differing amounts of time to draw a series of lines. The PRINT command is the fastest because its only function is to output ASCII characters. The additional functions performed by the DRAW command slow its execution. Whenever graphic information is output with the DRAW command, two additional functions must be performed: clipping and transformation from user data units to GDU's. RDRAW is slower still because it performs two more functions: rotation and computing absolute coordinates. (Only absolute graphic information is output. All relative calculations are completed inside the Graphic System prior to output.)

The above example program also demonstrates that PRINT handles arrays differently from DRAW and RDRAW. PRINT outputs data from an array in simple sequential order. The secondary address of 20 instructs the display to interpret the first ASCII number received

as a horizontal location in GDU's, the second ASCII number received as a corresponding vertical location in GDU's, the third number received as a new horizontal location, and so forth. When outputting data from arrays, DRAW and RDRAW require two data arrays. In the above example program, examine the DRAW command at statement 370. The location of the first displayed point is specified by X(1) and Y(1). The second point's location is specified by X(2) and Y(2).

The graphic commands DRAW, MOVE, and GIN each have a counterpart which addresses an actual display location.

These counterparts are listed in the following table:

Arguments interpreted as User Data Units	Arguments interpreted as GDU's
DRAW X,Y	PRINT @32,20: A,B
MOVE X,Y	PRINT @32,21: A,B
GIN X,Y	INPUT @32,24: A,B

For example, the execution of the statement PRINT @32,20: 65,50 always draws a line to the center of the screen regardless of the parameters of the WINDOW, VIEWPORT, and SCALE statements, and the present position of the graphic point. This command will draw a line to a point which is outside a specified window and viewport; the clipping and transformation done to lines produced by the DRAW command is not done to lines produced by the PRINT command. PRINT @32,21: A,B causes the graphic point to be moved to a point on the display specified by the contents of variables A and B. INPUT @32,24: A,B causes the actual location of the graphic point in GDU's to be placed in variables A and B. (This command and the GIN statement are two commands useful for determining where the graphic point is at a given time.) With all three of these statements, the first argument (called A in the above examples) is interpreted to be the horizontal location in GDU's, and the second argument (called B in the above examples) is interpreted to be the vertical location in GDU's.

There are no statements corresponding to RMOVE and RDRAW which operate in GDU's only.

Since graphic information is sent to any graphic device in the maximum precision of which the Graphic System is capable, the displayed resolution is determined by the graphic device hardware, not by the Graphic System. The number of addressable points on the Graphic System display is 1024 (horizontal) by 780 (vertical). Therefore, the display's resolution is approximately 1/8 or .125 GDU. With 3000 by 2000 addressable points, the 4662 plotter is significantly more precise. Its resolution is approximately .05 GDU.

The WINDOW and VIEWPORT commands determine how user data units are transformed to GDU's. This transformation process is entirely internal to the Graphic System and is totally independent of any graphic device, including the Graphic System display. Within the numeric limits of the Graphic System, any window can be transformed into any viewport. A command such as VIEWPORT 500,1000,200,400 will define a conceptually valid transformation which is used by the Graphic System. However, no graphic information will ever reach the display because all graphic information would be transformed into locations outside the 130 by 100 GDU limits of the Graphic System display. This condition is known as scissoring. It occurs when the graphic device hardware attempts to execute an action it is not capable of performing, such as moving the graphic point to the position 500,200 in GDU's. Scissoring usually reflects some kind of user error. What action actually results when scissoring occurs is determined by the hardware of the particular graphic device. Since graphic devices have different hardware configurations, scissoring produces different actions in different devices. When commanded to perform an action it is not capable of performing, the Graphic System display does nothing: the physical graphic point does not move.

Scissoring is very different from clipping, a needed function performed by the Graphic System before the graphic information is output to the graphic device. Clipping permits drawing to and from points located anywhere in user data space, even those which are outside the defined window.

Scissoring occurs when two types of errors have been made: when an improper viewport has been defined (e. g., VIEWPORT 300,500,200,400), and when the display is directly instructed to place the graphic point outside the GDU limits with a PRINT command (e. g., PRINT @32,20: 200,200).

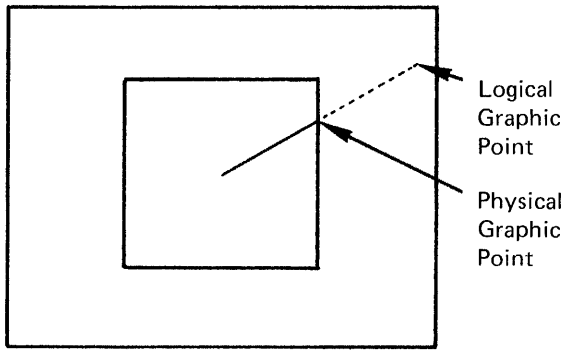
GRAPHIC POINT CONTROL

Throughout this manual, the words "cursor" and "graphic point" have been used almost interchangeably to specify a location on the display. There are situations where they do not represent the same location. It is sometimes necessary to distinguish between the location of the physical graphic point on the display and the location of the graphic point in user data space. When a program is not running, the location of the physical graphic point on the display is shown by the position of the blinking rectangular dot matrix. This is always the starting point for character information printed on the display. In the vast majority of instances, this is also the starting point for any lines drawn on the display.

The location of the graphic point in user data space is called the logical graphic point in this appendix. The position of the logical graphic point is affected only by graphic commands, such as MOVE and DRAW, not by other output commands such as PRINT. Whenever WINDOW, VIEWPORT, or SCALE are executed, the physical graphic point is not moved but the location of the logical graphic point (in user data space) is changed so that the two locations coincide. The physical and logical graphic points will not coincide after any of the following events occur:

- The execution of any non-graphic command which moves the physical graphic point on the display. PRINT is such a command. It changes the actual location of the graphic point without changing the location of the logical graphic point in user data space.

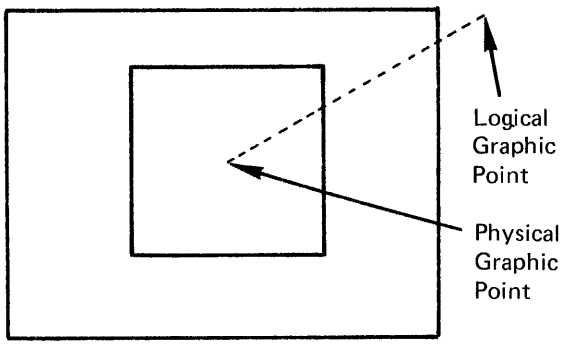
- The execution of a DRAW command to a point outside the defined window. Although the location of the logical graphic point is updated correctly, the physical graphic point is placed at the intersection of the drawn line and the defined window boundary.



All subsequent graphic commands are executed properly. However, the values returned by GIN will not reflect the location (in user data space) which coincides with the physical graphic point, not the logical graphic point.

* The execution of a MOVE command to a point whose transformed location is outside the GDU limits of the graphic device being addressed. In the case of the Graphic System display, this situation is produced by the following commands:

```
INIT
MOVE 200,200
```



When these commands are executed, the physical graphic point does not move at all, even though the logical graphic point is located correctly. As with the situation produced by a DRAW command described above, subsequent graphic command are executed properly but the GIN command will reflect the location of the physical graphic point, instead of the logical graphic point.

There are several ways to ensure that the physical and logical graphic points coincide. Here is one:

```
.
250 GIN A,B
260 MOVE A,B
.
```

The values returned by the GIN command show the location in user data space which coincides with the location of the physical graphic point. A MOVE to that point updates the location of the logical graphic point so that the two points coincide. This is useful when graphic and non-graphic information is mixed, as shown below:

```
.
310 GIN A,B
320 PRINT "HELLO";
330 MOVE A,B
.
```

Here is another method:

```
      .
      .
320 PRINT "HELLO"
330 RMOVE 0, 0
      .
      .
```

The RMOVE command at line 330 (above) will cause the physical and logical graphic points to coincide whenever the logical graphic point is inside the physical limits of the graphic device. This is another way to approach the problem solved on page 1-57.

TCS SUBROUTINES

Many Graphic System users may already be familiar with the Terminal Control System software of Tektronix. Listed below are TCS subroutines which are similar to Graphic System graphic commands:

TCS	GRAPHIC SYSTEM
DRAWA	DRAW
DRAWR	RDRAW
DRWABS	PRINT @32,20:
DWINDO	WINDOW
HOME	HOME
MOVE A	MOVE
MOVER	RMOVE
MOVABS	PRINT @32,21:
NEWPAG	PAGE
RROTAT	ROTATE
SEELOC	INPUT @32,24:
TWINDO	VIEWPORT
VCURSR	POINTER

No TCS subroutines are equivalent to the Graphic System commands GIN and SCALE. The RSCALE subroutine in TCS applies only to relative vectors while the SCALE command of the Graphic System applies to both absolute and relative lines. The TCS common variables TREALX and TREALY contain the same type of values which GIN places in its target variables: the location of the virtual cursor (or logical graphic point) in user data space. The Graphic System command AXIS has no TCS equivalent.

REFERENCES

Listed here are some materials which discuss computer graphics in general. Since much more has been published regarding specific topics in computer graphics, this is only a tiny fraction of the available literature.

BOOKS:

Newman, W. and Sproull, R., *Principles of Interactive Computer Graphics,* McGraw-Hill, NewYork, 1973. (Text book with detailed discussions of hardware, software, algorithms, data structures, and other topics. Includes large bibliography.)

Nelson, T. H., *Computer Lib/Dream Machines*, $7 from Hugo's Book Service, Box 2622, Chicago, Ill. 60690. (Informal introduction to computers in general. Also discusses computer graphics and other computer related topics.)

PERIODICALS:

Computer Graphics available from ACM-SIGGRAPH 1133 Avenue of the Americas, New York, New York 10036. SIGGRAPH is the special interest group for computer graphics, part of the Association for Computing Machinery. The Spring 1975 issue of Computer Graphics (Vol. 9 No. 1) is The Proceedings of The Second Annual Conference on Computer Graphics and Interactive Techniques.

ACM Computing Surveys Vol. 6, No. 1 (March 1974). ACM 1133 Avenue of the Americas, New York, New York 10036. This issue has two articles: "A Characterization of Ten Hidden Surface Algorithms" and "Computer Processing of Line Drawn Images".

Proceedings of The IEEE (April 1974). Institute of Electrical and Electronic Engineers, 347 East 47 Street, New York, New York 10017. Special issue on Computer Graphics.

ARTICLES OF GENERAL INTEREST:

Sutherland, I. E., "Computer Displays", *Scientific American*, June 1970, pp. 57-81.

CONFERENCE PROCEEDINGS:

Significant papers about computer graphics appear in proceedings from the National Computer Conference, The Fall Joint Computer Conference, and The Spring Joint Computer Conference. Available from:

AFIPS Press
210 Summit Avenue
Montvale, New Jersey 07645

ASCII CODE CHART

NUL (0)	DLE (16)	SP (32)	0 (48)	@ (64)	P (80)	\ (96)	p (112)
SOH (1)	DC1 (17)	! (33)	1 (49)	A (65)	Q (81)	a (97)	q (113)
STX (2)	DC2 (18)	" (34)	2 (50)	B (66)	R (82)	b (98)	r (114)
ETX (3)	DC3 (19)	# (35)	3 (51)	C (67)	S (83)	c (99)	s (115)
EOT (4)	DC4 (20)	$ (36)	4 (52)	D (68)	T (84)	d (100)	t (116)
ENQ (5)	NAK (21)	% (37)	5 (53)	E (69)	U (85)	e (101)	u (117)
ACK (6)	SYN (22)	& (38)	6 (54)	F (70)	V (86)	f (102)	v (118)
BEL (7)	ETB (23)	' (39)	7 (55)	G (71)	W (87)	g (103)	w (119)
BS (8)	CAN (24)	((40)	8 (56)	H (72)	X (88)	h (104)	x (120)
HT (9)	EM (25)	) (41)	9 (57)	I (73)	Y (89)	i (105)	y (121)
LF (10)	SUB (26)	* (42)	: (58)	J (74)	Z (90)	j (106)	z (122)
VT (11)	ESC (27)	+ (43)	; (59)	K (75)	[(91)	k (107)	{ (123)
FF (12)	FS (28)	, (44)	< (60)	L (76)	\ (92)	l (108)	¦ (124)
CR (13)	GS (29)	- (45)	= (61)	M (77)	] (93)	m (109)	} (125)
SO (14)	RS (30)	. (46)	> (62)	N (78)	^ (94)	n (110)	~ (126)
SI (15)	US (31)	/ (47)	? (63)	O (79)	_ (95)	o (111)	RUBOUT (DEL) (127)

Note

During string comparisons, lower case characters (gray tint) are converted to their upper case equivalents.

INDEX