

Additional Display Features

More on the -write- Command

It should be pointed out that the -at- command not only specifies a screen position for subsequent writing but also establishes a left margin for “carriage returns” (CR on the keyset), much like a typewriter. Upon completion of one line of text, the next line will start at the left margin set by the last -at- command. There are carriage returns implicit in “continued” write statements:

```
at      1215
write  Now is the
       time for all
       good men to
       come home.
```

The “at 1215” establishes a left margin at the 15th character position so that each line will start there. This example will produce an aligned screen display similar to the appearance of the tags of this continued -write- statement.

The setting of a margin by -at- has an unusual side effect. Consider:

```
at      2163
write  The cow jumped.
```

This will put the following display on the screen:

```
Th  
e  
co  
w  
ju  
mp  
ed  
.
```

This unusual display is caused by the setting of the left margin at character position 63, just two characters shy of the right edge of the screen. When a `-write-` would go past the right edge of the screen, TUTOR performs a carriage return to drop down one line, starting at the left margin. An `-arrow-` also sets a left margin with respect to the student typing a long response which would pass the right edge of the screen. Further typing appears on the next lower line starting at the margin set by `-arrow-`.

Occasionally, it is useful to position something on the screen without setting a margin. This can be done with an `-atnm-` command (“at with *no margin*”). The statement “`atnm 1215`”, is equivalent to “`at 1215`”, but does not change the current margin setting.

It is important to understand that writing characters on the screen automatically advances the terminal’s current screen position. Suppose we have consecutive `-write-` statements:

```
at    712  
write horses  
write and cows
```

This sequence will display “horseand cows” all on line 7. The first `-write-` (“horses”) advances the terminal’s screen position from the 712 specified by the preceding `-at-` to $712+6=718$ (there being 6 characters in the text “horses”). Without an explicit `-at-` to change this, the second `-write-` (“and cows”) starts at position 718. Note that:

```
at    712  
write horses  
and cows
```

would give a different display:

```
horses  
and cows
```

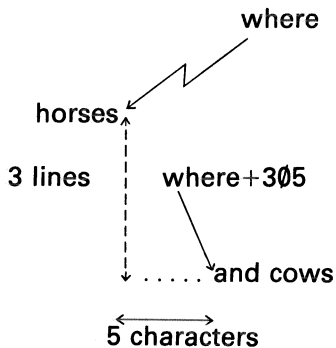
because the “continued” -write- statement implies carriage returns.

TUTOR keeps track of the current screen position in a system variable named “where”. For example:

```

at      712
write  horses
at      where+305  $$ “where” is 712+6=718 here
write  and cows
    
```

will produce the display:



The statement “write horses” leaves the screen position at 712+6=718, and the system variable “where” therefore has the value 718. When you then say “at where+305” this is equivalent to saying “at 718+305” or “at 1023”.

There are many uses of this “where” system variable. Here is another example:

```

at      1215
write  What is your name?
arrow  where+3
    
```

This will appear as:

What is your name? > Sam

The arrow has been positioned 3 characters beyond the end of the -write-statement’s display.

The positioning information is useful with other display commands as well. Consider this:

```

at      815
write  Look at this!
draw   where;815
    
```

This will display underlined text:

Look at this!

This is due to the fact that upon completion of the -write- statement, “where” refers to the beginning of the next character position just after the exclamation point. We simply draw from there back to the starting point. This form of the -draw- statement is so common that a concise form is permitted. For example, “draw ;815” is equivalent to “draw where;815”. Either form will draw a line or figure starting at the current screen position. This is particularly useful in constructing a graph (by connecting the new point to the last point with a line). The point reached with a -draw- (or *any* display command) will be the new screen position and may be referred to through the system variable “where”, which is kept up to date automatically by TUTOR.

There are fine-grid system variables “wherex” and “wherey” which correspond exactly to the coarse-grid “where”. The position “where+305” is equivalent to “wherex+(5×8),wherey-(3×16)” because a character space is 8 dots wide and 16 dots high. The minus sign is present because, in coarse grid, line 4 is *below* line 3, whereas in fine grid dot 472 is *above* dot 471.

Superscripts and subscripts may be typed either in a locking or nonlocking mode. To type “10²³” you can either: (a) press 1, press 0, press SUPER, press 2, press SUPER, press 3 (non-locking case); or (b) press 1, press 0, press shift-SUPER (that is, hold down the shift key while pressing SUPER), press 2, press 3. To get down from a locked superscript you type shift-SUB (locking subscript). Notice that in typing superscripts or subscripts the SUPER and SUB keys are pressed *and released* before typing the material to be moved up or down. You do *not* hold these keys down while typing, unlike the shift key used for making capital letters.

It is possible to overstrike characters to make combinations. The symbol “v” can be made by typing v, backspace, SUPER, minus sign. This will superimpose a raised minus sign above the v. The backspace is typed holding down the shift key while hitting the wide space bar at the bottom of the keyset. Similarly, “horse” can be typed by typing “horse” followed by five backspaces and five underline characters. Note that these superpositions of characters won’t work in “mode rewrite”, where a new character is written on the screen. In mode rewrite, the last example would show up as “_____”, the “horse” having been wiped out by the characters whose only visible dots are the low, horizontal bars.

Extensions to the Basic Character Set

We've seen examples of lower-case and upper-case characters, numbers, punctuation marks, superscripts, and subscripts. What if you need special accent marks, or an unusual mathematical symbol, or the entire Cyrillic alphabet for writing Russian? It is important that you be able to write text on the screen using the special symbols of your particular subject area. In addition, it is possible to use special characters to display small, intricate figures whose display would be slow and cumbersome if done with `-draw-` commands.

The PLATO terminal has 126 built-in characters (including those used so far) and storage for 126 additional characters which can be different in every lesson. For example, Russian lessons fill this additional character storage space with the Cyrillic alphabet, whereas there is a genetics lesson which fills the storage area with fruitfly parts which permit displaying flies by writing appropriate characters at appropriate positions on the screen. We will learn how to access all 252 characters (126 which are built-in and 126 which can be varied).

The 126 built-in characters include many useful symbols which do not appear on the keyset (since there aren't enough keys). This is due to the fact that the keys on the right of the keyset are reserved for various important functions (ERASE, BACK, STOP, etc.). In order to access the "hidden" characters it is necessary to first strike the ACCESS key (presently the shift- \square key) and then to strike a second key. Like SUPER and SUB, the ACCESS key is not held down but struck. You can press ACCESS, then "a" to get a Greek alpha; ACCESS-b for beta; ACCESS-m for mu; ACCESS= \neq for \neq ; and also ACCESS-<or> for \leq and \geq . It is useful to try ACCESS followed by every key (or shifted key) at a terminal to find approximately 36 useful hidden characters. In most cases, there is a mnemonic connection between the key which follows the ACCESS key and the hidden character which results, such as \neq being ACCESS= \neq . ACCESS followed by comma gives the symbol \uparrow mentioned in the discussion of the `-writec-` command in Chapter 6. ACCESS- \emptyset and ACCESS-1 give the symbols \langle and \rangle used for embedding `-show-` commands in `-write-` statements. (In the discussion of "micro tables" later in this chapter, we will see that the MICRO key is equivalent to the ACCESS key, under normal circumstances.)

You can get at the "alternate font" of 126 additional, modifiable characters by pressing the FONT key (the shifted MICRO key), then typing regular keys, which will produce characters from the alternate font. Which characters appear depends on what character set has been previously loaded into the terminal. The FONT key toggles you between the standard built-in font and the alternate font (you stay in the alternate

font until you strike FONT to return to the standard font). It is, therefore, not necessary to strike FONT for each symbol (unlike the way ACCESS works).

Here is an example of the use of a special character set:

```
at      912
write  Now LOADING CHARACTER SET.
       Please be patient - loading
       takes about 17 seconds.
charset charsets,russian
erase  $$ full-screen erase to remove message
unit   intro
at     905
write  The Russian word карандаш means pencil.
```

Fig. 9-1.

The -charset- statement sends to the terminal the character set specified in the tag (character set "charsets,russian" in this case). Character patterns are transmitted to the terminal at a rate of 7.5 character patterns per second, so a full 126-character set will take about 17 seconds to send. Precede the -charset- command with a -write- statement to explain this delay to the student, so that he or she will not think that something is wrong or broken! The full-screen -erase- will remove the message upon completion of the loading process. Once the character patterns have been

loaded into the terminal, it is possible to write Russian text on the student's screen at the same high speed as English, 180 characters per second, which corresponds to a reading speed of almost two thousand words per minute.

TUTOR keeps track of which character set has been loaded into the terminal and skips a `-charset-` statement if loading is not required. In the above example, TUTOR would rush right through the message, skipping the `-charset-` and erasing the screen. There would not be the 17-second delay which occurs if the Cyrillic characters have not been loaded.

The `-write-` statement in unit "intro" is created by:

1. typing "write The Russian word";
2. striking the FONT key to select the alternate font;
3. typing the keys k, a, r, a, n, d, a, w (which causes карандаш to appear)
4. striking the FONT key to toggle back to the standard font
5. typing " means pencil."

Each character in the alternate font is associated with a key on the keyset. For example, the creators of the "russian" character set chose to associate the Cyrillic "д" with the "d" key because of the phonetic similarity of these two letters. Similarly, the Cyrillic "р" and "н" sound like the "r" and "n" letters with whose keys they are associated. Just as accessing some of the 126 built-in characters requires the ACCESS key, so a full 126-character alternate font will also necessitate the use of the ACCESS key to reach some of the characters.

If the student is to respond at an `-arrow-` with a Russian response, he or she must hit the FONT key in order to do so. Usually it is preferable to precede the first judging command with the statement "force font", which essentially hits the FONT key for the student. The student merely uses the regular typing keys, but the typing appears in the alternate font. Some languages, including Arabic, Hebrew, and Persian, are written right-to-left instead of left-to-right. For these languages use a "force font,left" and the student's typing will automatically go leftwards from the `-arrow-` in the alternate font.

The "initial entry unit" (ieu)

You may have noticed that the first few statements of the previous example (which write a message, load a character set, and then erase the screen) are not preceded by a `-unit-` statement. This is intentional.

TUTOR statements which precede the first `-unit-` statement (“unit intro” in this case) constitute an “initial entry unit” which is performed whenever a student enters the lesson. The “initial entry unit” (or “ieu”) is the logical place to put various kinds of initializations, such as a `-charset-` statement to load characters which will be used throughout the lesson. Although `-define-`, `-vocabs-`, and `-list-` statements are not actually *executed* (they are only instructions to TUTOR on how to interpret `-calc-`, `-concept-`, and `-answer-` statements in preparing a lesson for student use), they can also be placed in the “ieu” at the beginning of the lesson, for the sake of readability.

The importance of the “ieu” lies in the fact that it is performed no matter where the student starts within the lesson (even if the student does not start at the first unit statement). TUTOR is capable of keeping track of a student’s place within a lesson, so that a student who leaves without finishing a lesson is able to restart the next day where he or she left off. It is important, in the restarting process, to load the appropriate character set. The restart procedure can *not* be executed properly if the `-charset-` statement comes after the first `-unit-` statement (since the student will not go through the first part of the lesson again).

Suppose the student is to restart in unit “middle”, which looks like this:

```
unit middle
next mid2
```

The “ieu” is utilized in such a way that TUTOR acts as though the “ieu” were done at the beginning of the restart unit:

```
unit middle
(do "ieu")
next mid2
```

This pseudo-do is the reason for following the `-charset-` statement with a full-screen erase. We don’t want the “loading” message to interfere with the display to be created by unit “middle”.


Smooth Animations Using Special Characters

The `-charset-` command is not limited to its use with foreign alphabets. Special characters are often used to create pictures:

```
at 1319
write This  uses special characters!
```


The car is composed of several adjacent characters. Because characters can be drawn very fast (180 per second), dramatic animations are possible:

```

mode  rewrite
do    drive,x←100,400
*
unit  drive
at    x,200
write 

```

The car advances one dot at a time. If the car characters are designed in such a way as to leave a vertical column of blank dots at the back of the car, the “rewrite” mode will insure that the advancing car simultaneously erases its old position. If two columns are left blank, the car could be advanced two dots at a time and still completely wipe out the previous car display. This type of animation can run as fast as twenty or thirty moves per second, which creates the illusion of a smoothly moving object.

For the built-in characters there is an expandable and rotatable (but slow) line-drawn form available through the use of `-size-` and `-rotate-`, but these commands have no effect on charset characters. If a larger or rotated car is needed, it can be constructed with `-draw-` and `-circle-` commands, built up out of additional special characters, or produced with “lineset” characters. A lineset is like a charset, but the characters are made up of lines instead of dots. If “size” is not zero, and a lineset is in effect, alternate-font text is displayed as line-drawn characters which can be expanded and rotated.

Creating a New Character Set

Figure 9-2 on the following page demonstrates how a special character is designed at a PLATO terminal. The author moves the cursor on an 8×16 grid to specify which dots are to be lit. The author can inspect “in the small” the appearance of the character he designs “in the large”. The letter shown at the top of the page is the key with which this character will be associated when typing in the alternate font, just as character “д” is associated with key “d” in “charset russian”. The character pattern is stored in such a way that the author can (at any later time) recall the pattern and modify it. A character set can contain up to 126 special characters or as few as one or two characters.

Your own character set will be stored in an electronic storage area assigned to you. Such storage areas are called “lesson spaces” because they mainly hold TUTOR statements which describe a lesson to be administered to students by PLATO. Your lesson space might be called “italian3” and it is by this name that you refer to the lesson space when you want to look at the TUTOR statements or change them. Within this lesson space you can also have one or more character sets, which you will have named. Suppose in lesson space “italian3” you have stored a character set named “rome”. In this case, the TUTOR statement used to transmit this character set to a terminal is:

```
charset italian3,rome
      ^           ^
      |           |
lesson space   character set
```

The same format holds for linesets.

Micro Tables

It is sometimes desirable to associate a string of several characters with a single key. For example, the symbol \bar{v} may be produced by `v`, `backspace`, `superscript`, `minus sign`. It is possible to set up a “micro table” so that \bar{v} may be produced simply by hitting the `MICRO` key followed by hitting “`v`”. Similarly, the micro table might specify that `MICRO-e` should be equivalent to typing `e`, `shift-SUPER`, `k`, `x`, `SUPER`, `2`, `shift-SUB` to make e^{kx^2} . The micro table makes possible a kind of shorthand which can be useful both to authors composing `-write-` statements and to students typing complicated responses.

Like character sets, micro tables reside in lesson spaces. If lesson space “italian3” contains a micro table named “dante”, these micros can be made available to students by the statement:

```
micro italian3,dante
```

As with `-charset-`, the `-micro-` statement should be placed in the “`ieu`” (initial entry unit).

Figure 9-4 on the following page shows how an author defines an item in a micro table, by associating a string of characters with a particular key. Later the effect of striking `MICRO` followed by this key is *identical* to typing this string of characters. With a “`force micro`” in effect, the student does not even have to press `MICRO`. This makes it easy to redefine the keyboard.

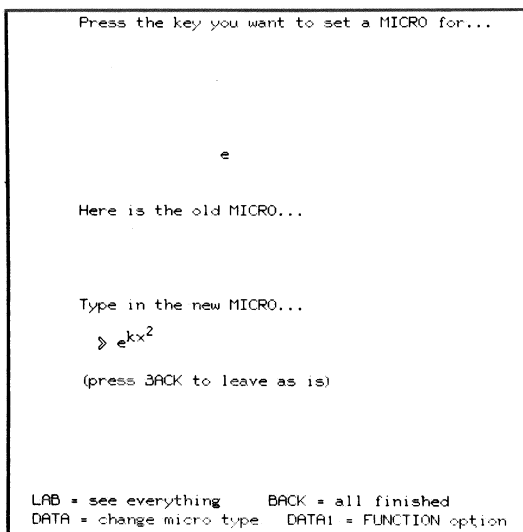


Fig. 9-4.

If you do not specify your own micro table, a standard one is provided that lets you use the MICRO key as though it were the ACCESS key. For example, MICRO-p gives ACCESS-p, which is π . This means you can (and should) mention only the MICRO key to students in your typing directions to them. It is not necessary to mention ACCESS. Note, however, that ACCESS-p must be used to make a π if you have your own micro table with a different definition for MICRO-p.

The Graphing Commands: Plotting Graphs with Scaling and Labeling

You may often want to plot a horizontal or vertical bar graph or other kinds of graphs to display relationships. There exists a group of TUTOR commands which *collectively* make it very easy to produce such displays. In particular, scaling of your variables to screen coordinates is automatic, as is the numerical labeling of the axes, with tick marks along the axes. Figure 9-5 shows some examples.

Suppose you want a graph to occupy the lower half of the screen. The horizontal x-axis should run from zero to ten and the vertical y-axis from zero to two. Both axes should be labeled appropriately. These statements will make the display shown in Figure 9-6.

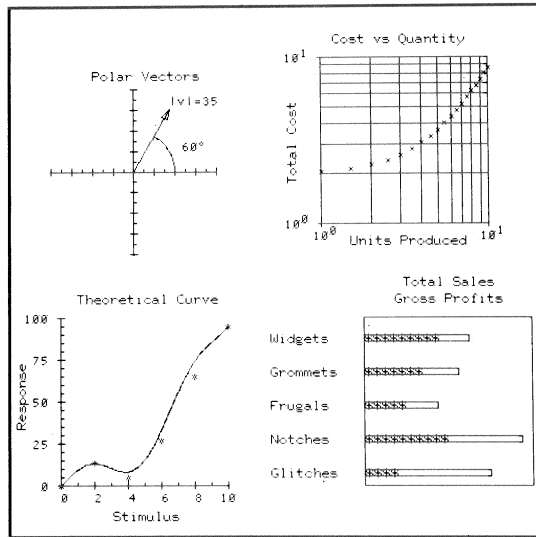


Fig. 9-5.

unit	setup	
gorigin	50,50	\$\$ x,y graph origin
axes	400,150	\$\$ lengths in dots
scalex	10	\$\$ maximum x
scaley	2	\$\$ maximum y
labelx	2,.5	\$\$ major mark every 2,
*		minor every .5
labely	.5	\$\$ major mark every .5
graph	6,1.5,A	\$\$ x=6, y=1.5
graph	8,.5,BC	\$\$ x=8, y=.5
hbar	3,1.5	\$\$ horizontal bar to
*		3,1.5
vbar	4.5,1	\$\$ vertical bar to
*		4.5,1
gdraw	2,.5;4,1.5;7,0	
gat	4,2	
write	Top	

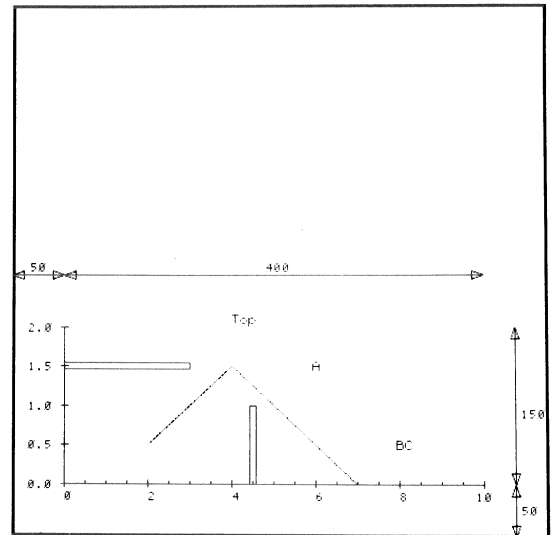


Fig. 9-6.

After specifying -gorigin- and -axes- in terms of fine-grid screen coordinates, the -scalex- and -scaley- commands associate scale values with the end points of the axes. These scale values determine how (x,y) coordinate positions given in later statements will be scaled to screen coordinates.

The `-labelx-` and `-labeley-` commands cause numerical labels and tick marks to appear. The statement “`graph 6,1.5,A`” plots an A at $x=6$, $y=1.5$ in scaled coordinates. The `-hbar-` and `-vbar-` commands draw horizontal and vertical bars to the specified scaled points. The `-gdraw-` command is like `-draw-`, except points are specified in terms of scaled quantities. The `-gat-` command is like `-at-` but uses scaled quantities.

Read the example over and try to identify in the picture what part of the display results from each statement. (Keep in mind that each number in the tags of these statements could have been a complicated mathematical expression.)

The `-markx-` and `-marky-` commands are similar to `-labelx-` and `-labeley-` but merely display tick marks without writing numerical labels. The `-axes-` command has an alternative form which allows for axes in the negative directions. (See Figure 9-7.)

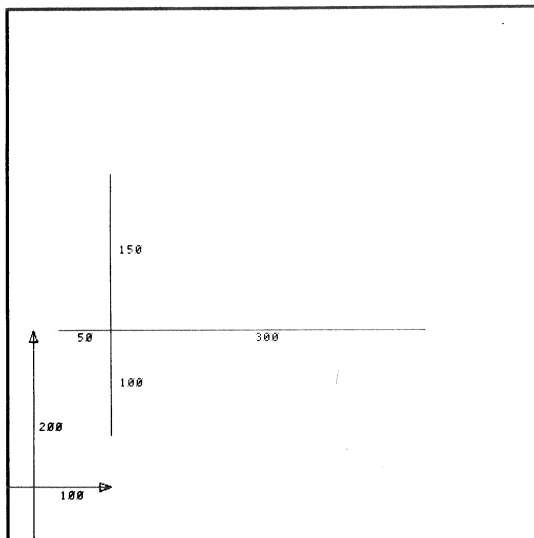
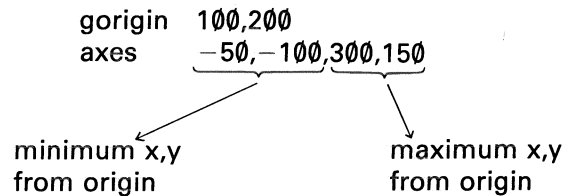


Fig. 9-7.



Although the commands were originally designed to make it easy to draw graphs, the automatic scaling features make these commands useful in many situations. Note, in particular, that you can move complicated displays around on the screen merely by changing the `-gorigin-` statement.

Additional graphing commands include `-gvector-` for drawing a line with an arrowhead at one end, `-polar-` for polar coordinates, and `-lscalex-` and `-lscaley-` for logarithmic scales. The `-bounds-` command has the same

effect as `-axes-` in establishing lengths, but no axes are drawn on the screen (a later blank `-axes-` command will display the axes). The `-gbox-` command is used to draw rectangular boxes easily. The `-gcircle-` command draws circles or, if the x- and y-scales are different, `-gcircle-` will draw an ellipse.

Functions can be plotted very easily with the `-funct-` command. For example, `"funct 5sin(2w),w<=1,5,.02"` will plot the function `"5sin(2w)"` by evaluating this function for values of `w` running from 1 to 5 in steps of `.02`. Note the similarity to the form of the iterative `-do-` statement. If there was an earlier `"delta .02"` statement, we can leave off the increment and simply write `"funct 5sin(2w),w<=1,5"`. If, in addition, we want the function to be plotted all the way from the left edge of the established axes to the right edge, we simply write `"funct 5sin(2w),w"`.

Summary of Line-drawing Commands: `-draw-`, `-gdraw-`, `-rdraw-`

Recall that the `-draw-` statement has the form:

```
draw point1;point2;point3;etc.
```

Each point in a `-draw-` statement may be coarse-grid (such as `"1215"`) or fine-grid (such as `"135,245"`). Each point specification is set off by a semicolon in order to avoid ambiguities when mixing coarse-grid and fine-grid points, as in `"draw 1525;1932;35,120;1525"` (the first two points are given in coarse-grid; the third, in fine-grid; and the last point in coarse-grid coordinates).

A discontinuous line drawing can be made with a single `-draw-` statement by using the word `"skip"`:

```
draw 1518;1538;skip;1738;1718
```

Using `"skip"` in a `-draw-` statement means `"skip to the next point without drawing a line."` This example is essentially equivalent to:

```
draw 1518;1538
draw 1738;1718
```

The only difference between these otherwise equivalent forms is related to the fact that the system variables `"where"`, `"wherex"`, and `"wherey"` are not brought up to date until the *completion* of the `-draw-` statement. The sequence:

```
at 1319 $$ affects "where"  
draw 1518;1538;skip;1738;where
```

is equivalent to:

```
at 1319  
draw 1518;1538  
draw 1738;1319
```

since *during* the -draw- statement "where" has the value 1319. On the other hand, the sequence:

```
at 1319  
draw 1518;1538  
draw 1738;where
```

is equivalent to:

```
at 1319  
draw 1518;1538  
draw 1738;1538
```

since upon completion of the first -draw- statement, the value of "where" is 1538. This difference between a single -draw- using "skip" and separate -draw- statements is sometimes useful in drawing figures relative to some point.

As mentioned earlier, starting with a semicolon implies a continued drawing from the present screen location. The sequence:

```
at 1319  
draw ;1542;1942
```

is equivalent to:

```
at 1319  
draw where;1542;1942
```

and is also equivalent to:

```
draw 1319;1542;1942
```

Sometimes you have more points for a -draw- than will fit on one line. A "continued" -draw- can be written, with the command blank on succeeding lines:


```
draw 1512;1542;skip;100,200;
      400,200;400,400;
      100,400;100,200
```

This will behave as though all the points had been listed on one line.

To summarize, the `-draw-` statement contains fine-grid or coarse-grid points separated by semicolons, “skip” can be used for a discontinuous drawing, “where” and the fine-grid “wherex” and “wherey” are brought up to date upon *completion* of the `-draw-`, and starting the tag with a semicolon has the special meaning of continuing a drawing from the present screen position.

The `-gdraw-` command is like the `-draw-` command except that points are relative to the graphing coordinate system established by `-gorigin-`, `-axes-`, (or `-bounds-`), `-scalex-`, and `-scaley-` (or logarithmic scales set up by `-lscalex-` and `-lscaley-`). Of particular value are the “skip” option and starting with a semicolon (for continuing a drawing). The use of “where”, “wherex”, and “wherey” in a `-gdraw-` statement is normally not meaningful, since these system variables refer to the absolute screen coordinate system, not the graphing system. In the graphing coordinate system, there are only fine-grid, not coarse-grid points, so all points have the form “x,y”.

It is possible to use `-draw-` to draw something relative to the present screen position:

```
at    2215
draw wherex+25,wherey-75;wherex+200,wherey+150
```

(Remember that “wherex” and “wherey” do not change until the completion of the `-draw-` statement.) There is an `-rdraw-` command (“r” for “relative”) which makes such drawings simpler. The example just shown can be written:

```
rorigin 2215
rdraw 25,-75;200,150
```

Each point of an `-rdraw-` is taken to be relative to an origin established with an `-rorigin-` command.

The `-rdraw-` command is particularly useful for applications such as writing the same Chinese characters at different places on the screen. For each character, make a subroutine involving one or more `-rdraw-` statements. The characters can be positioned with `-rorigin-` statements:

```
rorigin 400,400
do chin1
rorigin 400,300
do chin2
etc.
```

Or you might include the `-rorigin-` statement in the character subroutines:

```
do chin1(400,400)
do chin2(400,300)
```

In this case each subroutine has a form like this:

```
unit chin1(a,b)
rorigin a,b
rdraw -75,30;75,30;etc.
```

Unlike `-draw-`, the `-rdraw-` command is affected by preceding `-size-` and `-rotate-` commands. Your Chinese characters can be enlarged and rotated:

```
size 3,5 $$ 3 times the width, 5 times the height
rotate 45 $$ rotated 45 degrees
do chin1(400,400)
do chin2(400,300)
```

(Another way to handle such things as Chinese characters is with `-lineset-`.) Figure 9-8 shows a design created with the following commands:

```
rorigin 250,250
do figure,a←0,360,15
*
unit figure
rotate a
rdraw -50,0;50,0;0,200;-50,0
```

The `-rotate-` command affects `-rdraw-` even with “size 0”, even though `-write-` is *not* rotated in size 0. (The `-write-` statement is unaffected in order to facilitate normal text operations.) As far as `-rdraw-` is concerned, size 0 is equivalent to size 1. As far as `-write-` is concerned, size 0 means “write text at 180 characters per second, unrotated”, whereas size 1 means “write line-drawn text at 6 characters per second, rotated”.

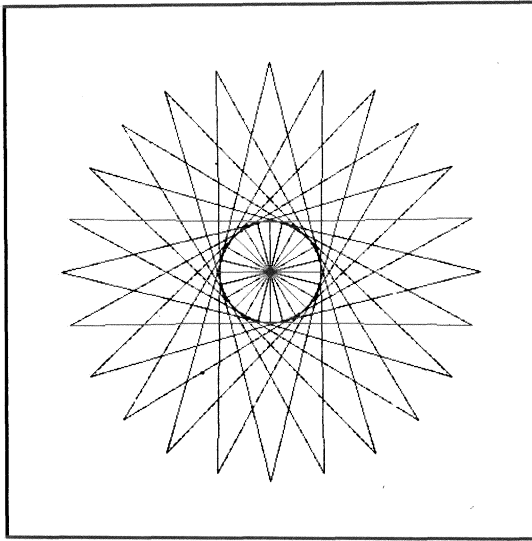


Fig. 9-8.

Note that `-rdraw-` and `-size-` are essentially reciprocal to `-gdraw-` and `-scalex-`. In the case of `-rdraw-`, a drawing gets bigger when `-size-` specifies a larger size. But, specifying a larger number in a `-scalex-` command implies that the same number of screen dots (given by `-axes-`) will now correspond to larger (scaled) numbers in a `-gdraw-`. This means that a *larger* `-scalex-` implies a *smaller* `-gdraw-` figure. Note that `-gorigin-` affects `-gdraw-` the same way that `-rorigin-` affects `-rdraw-`.

There is a complete set of “relative” commands for making displays relative to an origin specified by `-rorigin-`, and affected by `-size-` and `-rotate-`. Here is a summary:

“ABSOLUTE”	“RELATIVE” (-size-)	“GRAPHING” (-scalex-, -scaley-)
	<code>rorigin</code>	<code>gorigin</code>
<code>at</code>	<code>rat</code>	<code>gat</code>
<code>atnm</code>	<code>ratnm</code>	<code>gatnm</code>
<code>draw</code>	<code>rdraw</code>	<code>gdraw</code>
<code>box</code>	<code>rbox</code>	<code>gbox</code>
<code>vector</code>	<code>rvector</code>	<code>gvector</code>
<code>circle</code>	<code>rcircle</code>	<code>gcircle</code>

Note that `-rcircle-` will draw an ellipse if the x- and y-sizes are different (as in “size 1,4”, for example).

The “halfcirc” subroutine of Chapter 4 could be conveniently rewritten using relative commands:

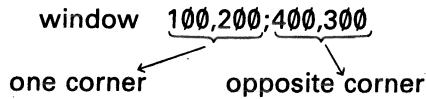
	“ABSOLUTE”		“RELATIVE”
unit	halfcirc	unit	halfcirc
at	x,y	rorigin	x,y \$\$ sets rorigin and “rat 0,0”
circle	radius,0,180	rcircle	radius,0,180
draw	x-radius,y;x+radius,y	rdraw	-radius,y;radius,y

It is important to note that the relative specifications set by -rorigin-, -size-, and -rotate-, as well as the graphing specifications set by -gorigin-, -bounds-, -scalex- (or -lscalex-) and -scaley- (or -lscaley-) *carry over* from one main unit to another. If you would prefer to have these parameters set to some standard values at the beginning of each main unit, simply do the initializations in an -imain- unit. (Remember that the -imain- command allows you to specify a unit to be performed every time a new main unit is started.)

How do you decide which of the three sets of display commands to use? If you want to rotate a drawing, you must use relative commands, because the absolute and graphing commands are unaffected by the -rotate- command. If rotations are not involved, just use whichever commands seem most convenient at the moment. Absolute commands may be used quite often since they are the simplest and easiest to use. The graphing commands are certainly best for drawing graphs of functions, but they are also useful whenever it is convenient to think of your drawing in terms of numerical scale factors. Graphing commands are also needed if you use polar coordinates (invoked with the -polar- command). Sometimes you may use all three sets simultaneously. For example, in one of this author’s lessons, the most convenient way to produce the screen display was to give instructions at the bottom of the screen using absolute commands, draw figures scaled in centimeters using graphing commands, and superimpose a movable box on the (absolute) instructions by means of relative commands.

The -window- Command

Sometimes it is useful to specify a “window” through which drawings are viewed. Parts of a figure extending outside the window are not drawn. A rectangular window is specified by giving the lower left and upper right corners of the desired window:



The corners could also be given in coarse-grid coordinates, as in “window 1524;1248”.

Drawings constructed from the various `-draw-` commands and `-circle-` commands are affected by a preceding `-window-` command. Line-drawn text (size non-zero) produced by `-write-`, `-writec-`, `-show-`, etc., will also be windowed. Like `-size-` and `-rotate-`, windowing is not reset upon entering a new main unit. Be sure to use a blank `-window-` command (blank tag) to turn off windowing operations. It is quite common for an author to forget to turn off windowing and then wonder why some of the drawings aren't showing up! The correct structure is shown below. (See Figures 9-9 and 9-10.)

```

window  one corner;opposite corner
.
.
.
(windowed) display statements
.
.
window          $$ blank tag to turn off
    
```

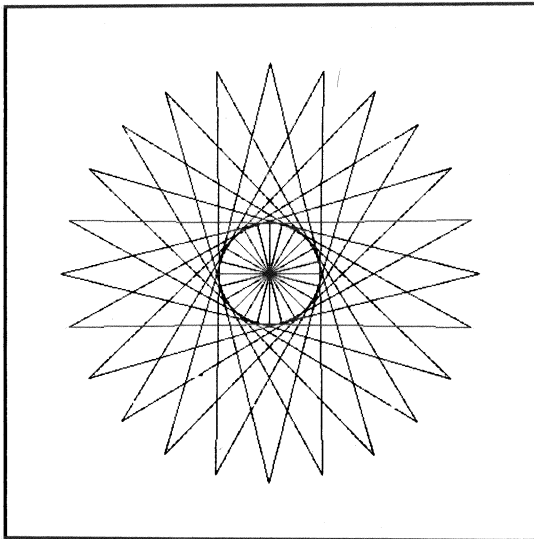


Fig. 9-9.

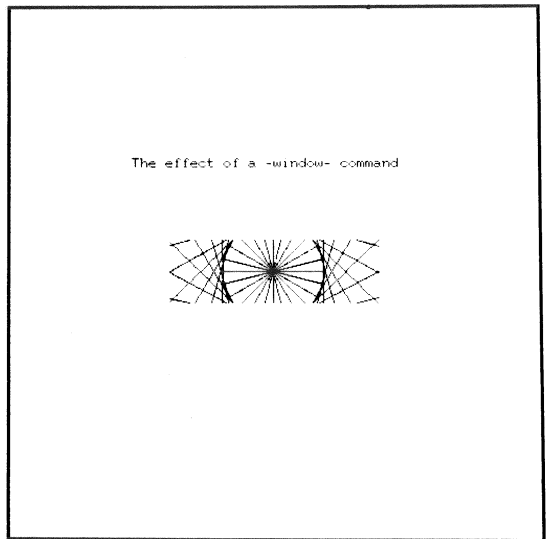


Fig. 9-10

More on Erasing: The -eraseu- Command

When a student's response is judged "no" or "wrong", he or she can correct the response by hitting ERASE or ERASE1 to erase a letter or word, or by hitting NEXT, EDIT, or EDIT1 to erase the entire response. If additional judging keys have been defined with a -jkey- command, these will act like NEXT and erase the response. If there is only one -arrow- command and no -endarrow-, these options are available even after an "ok" judgment (except that a NEXT key or another judging key takes the student to the next main unit rather than merely erasing the response). If there is a "force firsterase", the student need not clear an incorrect response by pressing NEXT before trying a different response. In this case, the first key of the new response will cause the old response to be erased.

If the student erases part or all of his or her response, the "ok" or "no" is erased. Moreover, the last response-contingent message to the student is erased, since it is no longer relevant. For example:

```
.  
.   
wrong  cat  
write  The cat is  
        not a canine.  
.   
.   
.
```

The student types "cat" and presses NEXT:

```
➤ cat no  
  
The cat is  
not a canine.
```

Notice that there is a default -at- three lines below the response. Suppose the student now presses ERASE:

```

> ca
    
```

The “t”, the “no”, and the text of the -write- statement have all disappeared automatically. This is appropriate since the comment “The cat is not a canine” is no longer needed.

It is helpful to know that the method TUTOR uses for automatically erasing such text is by re-executing the *last* -write-, -writec-, or -show-statement in the erase mode. Suppose we change the lesson slightly:

```

.
.
.
wrong cat
write The cat is
      not a canine.
write Meow!
.
.
    
```

Now the sequence looks like this:

```

> cat no

The cat is
not a canine. Meow!
    
```

```

> ca

The cat is
not a canine.
    
```

The TUTOR Language

Only the *last* -write- statement is removed, leaving “The cat is not a canine” on the screen. Notice that the normal automatic erasing can be prevented simply by adding an extra -write- statement. Even a blank -write- statement will do.

As another example, consider this:

```
.  
.   
.   
wrongv 4  
write   Number of apples=  
show    apnum  
.   
.   
.
```

Only the -show- will be erased, leaving “Number of apples=” on the screen. If this is not desirable, use an embedded -show-:

```
.  
.   
.   
wrongv 4  
write   Number of apples=<s,apnum>
```

Now the last -write- statement includes the showing of the number, and all the writing will be erased. It is important not to change “apnum” after the -write-. If you change its value from what it was when shown by the -write-, the re-execution in “mode erase” will turn off the wrong dots in the numerical part of the writing. Here is the type of sequence to be avoided:


```
.  
.   
.   
wrongv 4  
write   Number of apples=<s,apnum>  
calc    apnum←apnum+25  
.   
.   
.
```

The number will not be erased properly due to the change in “apnum”.

Similar problems can arise with the other `-show-` commands, including `-showa-`.

Sometimes the automatic erasing of the last text statement is insufficient. For example, if the reply to the student included a drawing produced with `-draw-`, or if there were several `-write-` statements, you would need some additional mechanism to remove the reply when the student presses ERASE. There is an `-eraseu-` command which you can use to specify a subroutine to be done when the student changes his or her response:

```

.
.
.
 eraseu  eblock
arrow   1215
.
.
.
unit    eblock
at      1512
erase   35,4
at      318
erase   42
.
.
.

```

Unit “`eblock`” will be done whenever the student changes a response. Only the *first* press of the ERASE key triggers the erase unit, since additional executions of the unit would be erasing nothing.

Another example involves an erase unit specific to a particular response:

```

.
.
.
wrong   3 dogs
do      woof
eraseu  remove
.
.
.

```

(Continued on the next page.)

```

unit    remove
mode    erase
do      woof
mode    write
eraseu
.
.
.

```

The statement “eraseu remove” defines unit “remove” as the unit to be done when the student presses ERASE (or NEXT, etc.). Unit “remove” in the example shown simply re-does unit “woof” in the erase mode, thus taking off the screen everything originally displayed by unit “woof”. The final blank -eraseu- clears the pointer so there is no longer an erase unit specified.

Notice the similarities between the -imain- and -eraseu- commands. Both specify units to be done under specific conditions.

Keeping Things on the Screen: “inhibit erase”

Let us consider a modified version of the simple language drill discussed in Chapter 7.

```

unit    espo
next    espo
back    satisfy
at      512
write   Here is a simple drill
        on the first five
        Esperanto numbers.
        Press BACK when you
        feel satisfied with your
        understanding.
at      1812
write   Give the Esperanto for
randu   item,5
at      2015
writec  item-2,one,two,three,four,five
arrow   2113
answerc item-2;unu;du;tri;kvar;kvin


```

This version will greatly annoy the student after the first couple questions. Each time the student gets an “ok” and presses NEXT to move on

to the next unit, the screen is erased and the student suffers through the introductory paragraph being written again on the screen. It turns out to be very annoying to see the same text replotted this way.

This is a situation where most of the material on the screen is not changing and should not be replotted. Only the item and the student's typing need be erased to make room for a new item and a new response. One way to do this involves judging correct responses "wrong", as was done in the dialog using -concept- discussed in Chapter 7. You should use "specs nookno" to prevent the "no" from appearing, or you can use the regular -okword- and -noword- commands to change the standard TUTOR "ok" and "no". For example, use the statement "noword Fine!" to cause "Fine!" to appear for a correct response. You would need to do a "noword no" whenever the student answers incorrectly. With all responses judged "wrong" we stay at the -arrow- and do not move on to another main unit.

Another way to manage a screen on which little is changing involves "inhibit erase". This statement prevents the normal full-screen erase upon leaving the present main unit. The next main unit must also execute an "inhibit erase" if no erase is to be performed upon leaving the second unit. We can rewrite our drill using this feature:

unit	preespo	
at	512	
write	Here is a simple drill	
	on the first five	
	Esperanto numbers.	
	Press BACK when you	
	feel satisfied with your	
	understanding.	
at	1812	
write	Give the Esperanto for	
goto	espo1	
*		
unit	espo	
at	2015	
erase	5	\$\$ item area
at	2115	
erase	15	\$\$ response area
entry	espo1	
 inhibit	erase	\$\$ leave instructions on screen
next	espo	
back	satisfy	

(Continued on next page.)

```

randu    item,5
at       2015
writec   item-2,one,two,three,four,five
arrow    2113
answerc  item-2;unu;du;tri;kvar;kvin

```

In unit “preespo” we display the instructions about the drill. We then go to “espo1”, where we “inhibit erase” and display the first item. After receiving an “ok”, the student moves on to the next main unit, “espo”. The screen is not erased since there was an “inhibit erase”. In unit “espo” we erase the area containing the displayed item, and we also erase the response area of the screen. We then fall through the -entry- command and display a new item. This process repeats continually, and only those parts of the screen which must be changed are erased.

It is important to place an explicit blank -erase- statement (“erase ”) at the beginning of unit “satisfy”. Since we have inhibited the normal full-screen erase, no erase will occur automatically when the student presses BACK to leave the drill. If unit “satisfy” does not explicitly erase the screen, the student will see a superposition of the drill display and the display produced by unit “satisfy”.

Similarly, if we specify a help unit, that unit should start with a full-screen erase. Upon completion of the help sequence, we should come back to unit “preespo” rather than “espo” in order to restore the screen display properly, like this:

```

.
.
.
entry  espo1
base   preespo  $$ to come back to preespo from help
help   esphelp
.
.
.

```

The -base- command puts us in a help sequence, with the base unit being “preespo”. When a base unit has already been specified, pressing HELP doesn’t change the base unit (in other words, there is only one “level” of help). When we reach an -end- command or press BACK, we will return to the base unit, which is preespo. Note that unit “satisfy” should have a blank base statement to insure that we are in a non-help sequence. Otherwise, pressing BACK in unit “satisfy” will bring us to the base unit “preespo” again.

Interaction of “inhibit erase” with -restart-

There is a -restart- command which is used to specify in which unit a student should resume study upon returning to a PLATO terminal. For example, suppose the last -restart- statement encountered on Monday by student “Ann North” in course “lingvo” was “restart espo” in lesson “espnun”. On Wednesday she returns to a PLATO terminal and identifies herself by name (Ann North) and course (lingvo). Her registration records will show that she is to be restarted in unit “espo” of lesson “espnun” and she will automatically be taken to that point. As discussed previously, the “ieu” (initial entry unit) will be done, which among other things permits character set loading.

Unfortunately, restarting at unit “espo” means that the basic drill instructions contained in unit “preespo” *will not appear* (see last example). This is basically an initialization problem. You should use -restart- commands in such a way as to restart students only at the *beginning* of a section of this kind. In this particular case, we should have had a “restart preespo” rather than “restart espo”. This is analogous to our use of “base preespo” for returning from a help sequence. (The more common form of the -restart- is the blank -restart-, which means “restart in the present main unit.” We would place a blank -restart- in unit “preespo”.)

Aside from initialization questions related to TUTOR and the display screen, it should be pointed out that the *student* has comparable initialization problems. Since the student may be away for several days, it is often advisable to have your restart points only at the beginning of sections of the lesson. This way the student can ease back into the context, whereas restarting in the middle of a discussion may be quite confusing. In those lessons which include an index, the index unit may be the best restart point. On the other hand, you will want to arrange things to allow the student to restart in the middle of a section if that section is very long.

When a student restarts in a lesson, he or she starts at the unit specified by the last -restart- command. However, the student’s saved variables, v1 through v150, have whatever values were current at the time he or she left the last PLATO class session. Therefore, some care is required to initialize appropriate variables in the restart unit.

The -char- and -plot- Commands

In most cases, special characters are handled with a -charset- command and displayed with a -write- statement using the FONT key. Alternatively, -char- commands can be used to transmit character patterns

to the terminal. If a `-char-` command sends a pattern to character slot 35 of the terminal, that character can be displayed using the `-plot-` command: `"plot 35"`. The arguments of the `-char-` command can be computed expressions so that a character can be constructed algorithmically. Similarly, the `-plot-` command may have a mathematical expression for its tag in order to choose the Nth character. See Appendix A for sources of detailed information on the `-char-` command.

The `-dot-` Command

The statement `"dot 125,375"` will plot a single dot at the specified location (`"dot 1817"` uses coarse grid). A sequence of `-dot-` commands can produce sixty dots per second on the plasma display panel. A `-draw-` with one point (`"draw 125,375"` or `"draw 1817"`) makes a single dot by drawing a minute line from this point to the same point (or itself) and, for technical reasons, will produce only twenty dots per second. (The commands `-rdot-` and `-gdot-` also exist.)