

PLATO[®] Author Language: Part II

A course in the
CREATE curriculum

Third Edition

Pub. No. 76360727B

Copyright © 1978 by Control Data Corporation

All rights reserved. No part of this material may be reproduced by any means without permission in writing from the publisher.

Printed in the United States of America.

1 2 3 4 5 6 7 8 9/84 83 82 81 80

Acknowledgments

The Control Data Education Company wishes to acknowledge the comments and contributions of the following consultants:

Daniel E. Bailey, University of Colorado; K. Dean Black, Brigham Young University; Karen K. Block, Learning Research and Development Center, University of Pittsburgh; C. Victor Bunderson, Brigham Young University; D. Cecil Clark, Brigham Young University; Celia R. Davis, University of Illinois; Walter Dick, Florida State University; Robert Glaser, Learning Research and Development Center, University of Pittsburgh; Keith A. Hall, Pennsylvania State University; Joseph A. Klecka, University of Illinois; Jerry W. Klein, West Virginia College of Graduate Studies; Helen Koch, Minnesota Educational Computing Consortium; M. David Merrill, Brigham Young University; Gaylene R. Rosaschi, Brigham Young University; Daniel L. Stufflebeam, Western Michigan University; Karl L. Zinn, University of Michigan; and The Center for Educational Design at Florida State University, which contributed significantly to portions of an earlier edition of this curriculum.

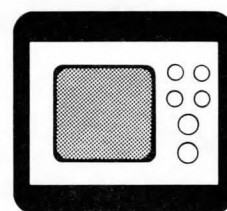
Visual Cues in *CREATE*

These visual cues at the beginning of each learning activity in the *CREATE* curriculum indicate the media used for instruction:

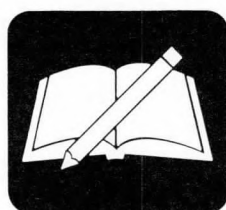
TEXT



VIDEOTAPE



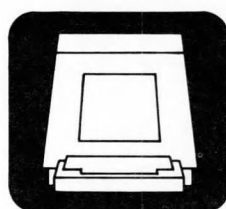
**TEXT
AND
EXERCISE**



FILMSTRIP



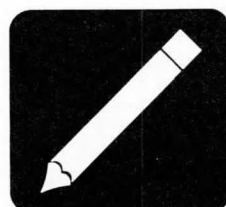
**COMPUTER-
ASSISTED
INSTRUCTION**



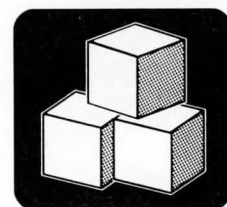
AUDIOTAPE



EXERCISE



PROJECT



Contents

PLATO AUTHOR LANGUAGE: PART II COURSE INTRODUCTION 1

Unit 1. Character Creation, Animation, and Display Generation 3

- 1-A. Character Set Creation 5
- 1-B. Character Set Creation: Additional Options 9
- 1-C. Animation 14
- 1-D. More Automated Displays 19

Unit 2. Relocatable Graphics and Graphs 27

- 2-A. Relocatable Graphics 29
- 2-B. Sizing and Rotating Relocatable Graphics 37
- 2-C. Creating Scaled Relocatable Graphs 40
- 2-D. Filling in Your Relocatable Graph 53

Unit 3. Variable Manipulation 65

- 3-A. Numbering Systems 67
- 3-B. Variable Formats 78
- 3-C. Segmented Variables 83
- 3-D. Stretching Your Variables 91
- 3-E. Author-Defined Arrays 98
- 3-F. System-Defined and Author-Defined Functions 106
- 3-G. Common and Storage Variables 111
- 3-H. Datasets and Namesets 120
- 3-I. Argument Passing: Another Way to Assign Values 123

Unit 4. Response Processing Options 126

- 4-A. More About Arrow Processing: Regular Instructions 128
- 4-B. The -judge- Instruction 135
- 4-C. More About Arrow Processing: Judging Instructions 140
- 4-D. More About the -specs- Instruction 144
- 4-E. Logical Operations 147
- 4-F. The -storen- Instruction 150
- 4-G. Application Program 153

Unit 5. Pause and Touch Panel Processing	156
5-A. The Touch Panel	158
5-B. Pause Processing with Tough	176
5-C. A Cumulative Program	184
Unit 6. Lesson Control	191
6-A. Connecting Lessons with the -jumpout- Instruction	193
6-B. Routers	197

PLATO Author Language: Part II

Course Introduction

The PLATO programmer has to be ready to try to program any lesson a CAI designer/developer dreams up. Luckily, the PLATO Author Language is extensive; many instructions exist, and still more will be incorporated into the system as time passes. Depending on your situation, you will use some instructions more often than others. Neither you nor the designers of this course, however, can foresee all the instructions you will require in your programming.

The instructions presented in *PLATO Author Language: Part II* are those the designers feel you are likely to require at some time in your programming career. You may not need all of them in the near future, but when you do, you will have them at your command.

The designers also feel that as a programmer, you must be able to sort through the instructions you know in order to choose the best instructions for a situation. Some of the instructions presented in this course do not have one specific purpose; in fact, most have more purposes than are mentioned. You must let your experience and imagination help you to realize each instruction's full value.

You may find some of the programmable-ready lessons in this course a little more difficult to program than those presented in *PLATO Author Language: Part I*. They are usually shorter, but may take just as much time to program as longer, easier lessons. Accept them as a challenge. Experiment. Try to find a combination of instructions that produces the desired results in the most efficient way. But don't worry; programming hints are often provided, and acceptable solution code is always provided to help you out.

On the other hand, some of the information presented may not be as detailed as you would like. At this point, however, once you know that a capability exists, you have tools available to help you learn more about it. You are encouraged to

consult *aids*, your reference manual, and the system consultants whenever you need to learn more about an instruction or capability.

This course contains six units:

Unit 1, "Character Creation, Animation, and Display Generation," explains how to create and animate single and multiple characters. It also explains ID/SD options not discussed in *PLATO Author Language: Part I*.

Unit 2, "Relocatable Graphics and Graphs," presents instructions that allow you to relocate, size, and rotate graphics and graphs.

Unit 3, "Variable Manipulation," discusses numerous ways to add to and make better use of your variable storage space.

Unit 4, "Response Processing Options," presents response processing options beyond those contained in *PLATO Author Language: Part I*.

Unit 5, "Pause and Touch Panel Processing," explains instructions that allow students to respond by touching the PLATO screen.

Unit 6, "Lesson Control," explains how to connect lessons with the -jumpout-instruction and presents general information on routers.

1

Character Creation, Animation, and Display Generation

This unit discusses the creation of more sophisticated graphics; this is accomplished by designing single and multiple characters in a character set block. It explains how to make characters “come to life” with the animation processes. It also presents more ID/SD options. You will create circles, broken circles, arcs, broken arcs, arrows, boxes, vectors, and broken lines and have them inserted in your code automatically. You will also learn to insert other instructions not automatically inserted in ID/SD.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify steps in the process of creating and loading single characters
- Identify steps in the processes of accessing lists of keys that have and have not been assigned to characters; creating multiple characters; copying characters from another character set; and copying characters for modification purposes.
- Identify steps in the processes of continuous motion and leaping motion character animation
- Identify steps required to use ID/SD to create circles and arcs; create broken circles and broken arcs; insert -arrow- instructions; create boxes, vectors, and broken lines; and insert other instructions not automatically inserted

LEARNING ACTIVITIES

- _____ 1-A. Character Set Creation. CAI: This activity explains how to design and insert your own characters. It also presents step-by-step practice in single character creation.
- _____ 1-B. Character Set Creation: Additional Options. Exercise/CAI: This activity explains options that allow you to access lists of keys you have and have not used for character creation; create multiple characters; copy characters from other character sets; and modify characters. It also provides step-by-step practice in using these options.
- _____ 1-C. Animation. Text/Exercise/CAI: This activity explains how to animate characters using either continuous or leaping motions. Practice in animation is provided.
- _____ 1-D. More Automated Displays. Text/Exercise/CAI: This activity steps you through the processes for creating circles, broken circles, arcs, broken arcs, and -arrow- instructions in ID/SD. It also steps you through processes for automatically creating boxes, vectors, and broken lines. Finally, you are shown how to insert other instructions not automatically inserted in ID/SD.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the processes of character creation, animation, and display generation, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

1-A. CHARACTER SET CREATION



Identify steps in the process of creating and loading single characters.

In *PLATO Author Language: Part I*, you learned to create basic geometric shapes. There may be times, however, when you wish to create special characters that require more sophisticated graphics. This activity discusses instructions used to create special characters and allows you to practice designing characters using these instructions.

CAI INTRODUCTION

Now sign on to a Control Data PLATO terminal using the student sign-on. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "Introduction to Character Sets: Build Your Own Characters," press LAB to take the lesson.

SUMMARY

A character space is eight dots wide and sixteen dots high. Each character space has more dots than are required by a letter of the alphabet. Characters spaces can be used to create shapes by specifying dots to be displayed.

To create characters, you must access a special type of block. You create this block for your special characters by:

1. Pressing a shifted letter on the block display to create a new block
2. Selecting the option to create a charset (character set) block
3. Naming the block

After choosing a name, you will be taken to a display that offers many editing options. You will probably begin with option 3.

The following function keys are available on the index display:

- HELP if you need help
- NEXT for normal design

- DATA for octal design
- SHIFT-HELP to delete

NEXT is the easiest key to use for creation of special characters.

Every character you create must directly correspond to a key on the keyboard. After you have pressed number 3, you will see a display requesting that key.

The lines on the character design area correspond to the dimensions of PLATO capital and small letters. Upon request you can see what the character you have created looks like.

The + on the editing squares of the design area is called the cursor. The cursor shows you the dot in which you are working. The arrow keys move the cursor; if you move the cursor off the editing board, it will reappear on the opposite side of the board.

The mode keys (+, o, -) are used to create your characters. The *i*, *B*, and *F* keys speed up editing.

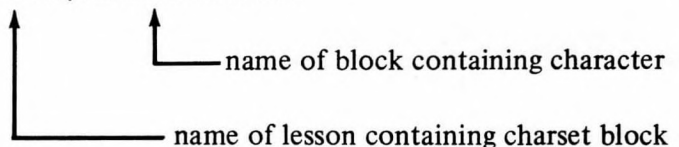
If you are creating dots, you are in "storing" mode; if you are erasing dots, you are in "removing" mode. Once a mode has been set, you remain in this mode until you press another mode key.

The following keys can be used to create a character:

- *F* fills in the entire character space
- *B* blanks the entire character space
- *o* puts a dot in a particular square of your design area (store mode)
- *i* inspects the character
- + allows you to move over squares without changing what is in the squares (travel mode)
- - allows you to remove dots (remove mode)

To use the character you have created, go to the correct line of your code. Type *write* and press the TAB key. Press the FONT key and the letter of the key that corresponds to the character you are inserting. To insert normal characters after inserting a special character, press the FONT key again. To load characters into your terminal, use the -charset- instruction. It should precede the first unit of your lesson. The format is:

```
charset CHARSETLESSON,CHARSETBLOCK
```



Because it takes up to seventeen seconds to load a character set, it is best to inform the students of the delay.

The `-chartst-` instruction allows you to see if the character set specified is already in the terminal. It has the same format as `-charset-` instruction:

```
chartst CHARSETLESSON,CHARSETBLOCK
```

After either the `-charset-` or the `-chartst-` instruction has been executed, the PLATO system puts a value into the system variable `zreturn`. If the `-charset-` load is successful, `zreturn` is set to `-1`. If the load is unsuccessful, `zreturn` will be set to a value other than `-1`, depending on the error condition.

The `zreturn` variable can be used in many ways. This code demonstrates one use:

```
chartst lesson,yours
branch zreturn,ldone,x
at      1115
write   Setting up lesson
charset lesson,yours
erase
ldone
.
.
.
```

With this code, if the `-chartst-` instruction causes `zreturn` to be set to `-1` (if the characters have already been loaded), the student branches to `ldone`; the “setting up lesson” display and the `-charset-` instruction are skipped because they are unnecessary. If `zreturn` is set to any other value, meaning that the loading has *not* been performed, the `-at-`, `-write-`, and `-charset-` instructions are executed.

CHECK YOUR UNDERSTANDING: CHARACTER CREATION

In this section of this activity, you will create two single characters, load your characters, and insert them in your code.

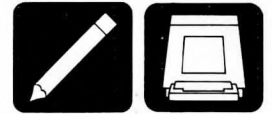
1. Sign on to a PLATO terminal as an author.
2. Go to your block display.
3. Create a charset block (press a shifted letter, press the number 2, and name the block).
4. You should now see the options index. Type the number 3.
5. Create a single character, following the directions on the displays.

6. Press BACK. Type in an unused letter.
7. Create another single character.
8. Press BACK until you return to the block display.
9. Include this set of loading instructions in your IEU:

```
chartst YOUR LESSON, YOUR CHARSET BLOCK
branch  zreturn, ldone, x
at      1115
write   Setting up lesson...
charset YOUR LESSON, YOUR CHARSET BLOCK
erase
ldone
.
.
.
```

10. Go into an empty block. Prepare to insert.
11. Type a -unit- instruction. Type an -at- instruction to position your first character. Press NEXT.
12. Type *write*. Press the TAB key. Press the FONT key. Type the letter corresponding to the first character you created.
13. Press NEXT. Type an -at- instruction to position your second character.
14. Insert your second character the same way you inserted your first character.
15. Condense your code. You should now see your characters as a student would see them.

1-B. CHARACTER SET CREATION: ADDITIONAL OPTIONS



Identify steps in the processes of accessing lists of keys that have and have not been assigned to characters; creating multiple characters; copying characters from another character set; and copying characters for modification purposes.

Activity 1-A discusses the creation of single characters. Often, however, you will want to combine characters to create a larger design, or “copy” characters from the *aids* picture library for use in your lessons. This activity steps you through these and other character creation options. You will discover how to:

- Access a list that shows you which keys you have used to correspond to the characters you have created
- Access a list that shows you which keys have *not* been used
- Create multiple characters (combinations of characters)
- Copy characters from another character set (such as the *aids* picture library)
- Copy a character from your own character set in order to modify that character

Sign on to a PLATO terminal as an author to begin this exercise.

THE KEYS USED LIST

When you have used a number of characters in a lesson, you may have difficulty remembering which keys you have already used to correspond to your characters. In this section of this exercise, you will access a list that shows you what keys you have used, and the characters that correspond to those keys.

- 1a. Go to your block display. If you have not previously created a charset (special character set) block, do so now. Press the capital letter of your last block, press number 2 for a charset block, and name the block.
- 1b. If you *have* created a charset block, at the block display, press the letter of your charset block (the letters *chars* appear to the left of this letter). When you see the display that begins with *charset block (your block)*, press NEXT to edit.
2. Now you should see an options index. Type the number of the option entitled *Memory slots used*. If you created characters in the last activity, you will see a list of the characters and their corresponding keys. If you did

not create a character in your *own* lesson space, the list will consist of those keys not allowed by the system.

3. Press BACK to return to the options index.

THE KEYS AVAILABLE LIST

This section of this exercise shows you how to access a list that will show you which keys you have not yet assigned to special characters.

1. Type the number of the option entitled *memory Slots available*. You should now see a list of keys to which you can assign characters (empty or available keys.) Do *not* press any keys at this time.
2. Press BACK to return to the options index.

MULTIPLE CHARACTER CREATION

When you want to combine characters to create a larger picture or display, you can use the multiple character option; this is much easier than trying to combine single characters. This section of this exercise shows you how to create multiple characters.

1. Type the number of the option entitled *Multiple character add/inspect/modify*.
2. You will see a grid of large squares. Each square represents a character. The arrow in the upper left-hand square is waiting for a character key to be assigned (the key that will correspond to that character). Type in a key you have not yet used to create a character. The key will appear in the square.
3. Notice the instructions at the bottom of the display. They tell you how to move the arrow. When you press a key to assign a letter to a character, the arrow automatically moves one square to the right. Your arrow should now be in the second square from the left. Type a key in this square now and *stop*. (When creating multiple characters, it is best to choose a group or pattern of keys that you can easily remember.)
4. Type in a character key in the third square. Write down the letters of your three keys here. _____
(keys)
5. Press the SUB key to move down a square.
6. Press the BACK key three times to move back three squares.
7. Type in character keys for the first three squares in the second line. These are the squares you will work with in this section of this exercise. Record the keys you used here. _____
(keys)
8. Press DATA.

9. You should see a large grid with a cursor in the upper left-hand corner. This cursor is used to create multiple characters; the same keys are used for multiple character creation as are used for single character creation except that *s* as well as *o* will store the dots you turn on. If you have a touch panel option on your terminal, you can move the cursor with your finger if you are in travel mode. If you have a touch panel, move the cursor by touch several times.
10. The first three large squares outlined in bold lines in the upper left-hand corner of this grid and the first three large squares of the next line correspond to the six squares to which you assigned keys on the previous grid. Use the character creation options (listed in the right-hand corner) to create a character using these six squares. When you finish, press SHIFT-BACK.
11. You could now create another character. At this time, however, press SHIFT-BACK to return to the options index.
12. Press BACK until you get to the block display. You are now going to put your characters into a section of your lesson code.
13. Press the letter of a block in which you can insert code.
14. Enter insert mode to insert after your last line of code.
15. Type an -at- instruction to position your characters on the screen. Press NEXT.
16. Type *write*, and then press the TAB key.
17. Press the FONT key (shifted MICRO). Type the three letters you chose for the squares on the top line of your grid.
18. Press NEXT to move to the next line. Press the TAB key.
19. Press the FONT key and the three letters you chose for the second line of your grid.
20. Press the BACK key. Your multiple character is now included in your code. (You will have to include a -charset- instruction in your code if you have not already done so to enable the PLATO system to load your character.)
21. Condense your code. You should now see your new multiple character on the screen as the student would see it.

COPYING CHARACTERS FROM ANOTHER CHARACTER SET

Many times you will want to use a picture that is available in the *aids* picture library. In this section of this exercise, you will select a picture from the *aids* library and copy it into your own character set.

1. Go to the author mode display. Access the *aids* lesson (SHIFT-a or type *aids* and press NEXT). Press DATA. Type *pictures* and press NEXT.
2. Press NEXT until you find a picture you like (choose a small one).

3. When you find a picture, write down the information at the top of your screen, such as *pictures*, *animals b*, just as it appears on your screen. You will need this information to copy your character.

(record information here)

4. Now write down the keys that appear by your picture just as you see them.

keys

5. Press SHIFT-NEXT to leave *aids*.
6. Access your own file space. Go to the block display. Type the letter of your charset block, and press NEXT to edit.
7. Type the number of the option entitled, *Copy from another charset*.
8. Type the lesson name you wrote down in *aids* (pictures). Press NEXT.
9. Type the charset name (*animals b* for example). Press NEXT.
10. Press NEXT to copy this character.
11. When you see the copy-a-character display, type the first key of the character you wish to copy, as it appeared in *aids*.
12. When the arrow moves to *To character*, type a key available in your charset.
13. Continue in this manner until you have assigned all the picture keys you wrote down in *aids* to empty keys in your charset. Record the new keys you used here.

(keys)

14. Press BACK until you get to your block display.
15. Now you will put the character you copied into your lesson code. Press the letter of a block in which you have space.
16. Prepare to insert after the last line of code.
17. Code an -at- instruction. Press NEXT.
18. Type *write* and press the TAB key.
19. Press the FONT key. Type in the first line of keys for the picture you copied. Be sure to use the keys assigned from your own charset.
20. Press NEXT, press the TAB key, press the FONT key, and type in the second line of characters.

21. When you have typed in all the keys for all the lines associated with your copied character, press BACK. Your character is now in your code.
22. Condense your code to see the character as a student would see it.

MAKING MORE COPIES FOR MODIFICATION

There may be times when you wish to use the same character many times with slight modifications. This section of this exercise explains these options.

1. Return to your block display. Press the letter of your charset block. Press NEXT.
2. Press the letter of the option entitled *Make another copy of a character*.
3. Type in the first letter (from your own charset) of the character you coded in the last section of this exercise. When the arrow moves to *To character*, type in the letter of an unused key. Continue until you have copied the entire character. Write the new letters you chose here. _____
4. Return to the options list. Choose the option entitled *Multiple character add/inspect/modify*.
5. In the grid of large squares, type the letters you have just assigned.
6. Press DATA.
7. You can now modify that character in any way you wish; thus, you can use much of your original character, yet save the work of redrawing the entire character to make a slight modification.
8. Press SHIFT-HELP. If you have modified your character, you will probably want to answer *no* to this question; otherwise, your character will be deleted. Press *n* now.
9. Press SHIFT-BACK.
10. Press SHIFT-BACK again for the index.
11. Press BACK until you get to your block display.
12. Put your new character in your code, using the new letters you chose.
13. Condense your code to see the character as a student.
14. At this time, you may either experiment further with the character creation options, or you may sign off the terminal and begin the next activity.



1-C. ANIMATION

Identify steps in the processes of continuing motion and leaping motion character animation.

Once you know how to create characters, you can make your characters “come to life”; you can move them across the screen. This activity discusses two ways to animate characters. One method moves the character in a continuous line. The other method moves the character by leaps.

CONTINUOUS MOTION

Before you animate your character, you should understand three forms of the -mode- instruction:

```
mode    write
mode    erase
mode    rewrite
```

The -mode- command with a *write* tag writes in dots. This is the standard mode, and it is set by default when you begin programming; the text you have displayed thus far has been displayed in mode write.

The -mode- command with an *erase* tag turns dots *off*. If you displayed a sentence in mode write, and rewrote the same sentence in the same place in mode erase, the sentence would be “erased” because the dots would be turned off. Whenever you include a -mode- instruction with an *erase* tag, you must remember to return to mode write or mode rewrite when you want to stop erasing; otherwise, your text will be invisible.

The -mode- command with a *rewrite* tag erases the old characters and replaces them with new ones very quickly. It has the same effect as erasing a character with an -erase- instruction and then writing a new character with a -write- instruction, but it performs the process much faster.

To animate a character, you simply rewrite the character over and over, positioning the character one or more dots over, back, up or down each time you rewrite. The more dots you move with each rewrite, the “faster” your character moves.

For each dot you want to move in your rewrite, you must leave one column or row of dots blank when you create your character to avoid leaving a *trail*; leave blanks on the side opposite the direction your character is moving. The leftmost column(s) should be blank if your character is moving to the right; the bottom row(s) should be blank if your character is moving up, and so forth. You will see what is meant by *trail* when you complete the practice section of this activity. Another way to avoid leaving a trail is to leave a blank space in your code on the side opposite the direction your character is moving.

Animation is accomplished easily with a -doto- loop:

```
define index = v1
unit move
mode rewrite
doto 2far, index+50, 350
at index, 50
write YOUR CHARACTER
2far
```

In this code, the variable *index* is used in the -at- instruction to set the position of the character. The variable *index* is incremented by 1 with each execution of the loop, thus moving the character one dot over to the right until the character halts at position 350.

If you want the character to move two dots at a time, add an argument to the -doto- instruction:

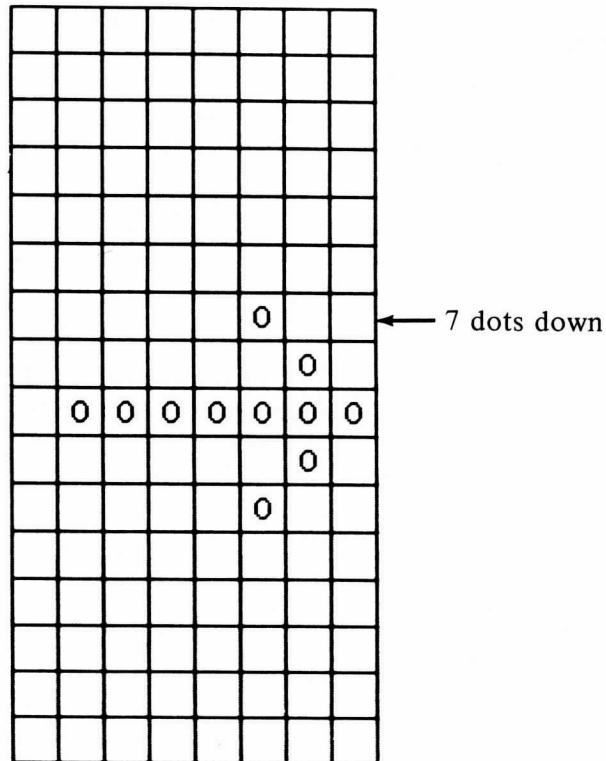
```
doto 2far, index+50, 350, 2
```

To move three dots at a time, use an argument of 3, and so on. Remember that if you move in increments of 2, 3, or more, you must leave the corresponding number of blank columns when you create your character.

CHECK YOUR UNDERSTANDING: CONTINUOUS MOTION

In this section, you will enter code to animate a character.

1. Sign on to the PLATO terminal as an author.
2. Create a character set block.
3. Create this single character:



This is the character you will animate. Notice that the leftmost column is blank; you will move the character to the right, one dot at a time.

4. Return to the block display.
5. Enter an empty block and enter this code:

```
define index = v1
unit motion
mode rewrite
doto 1stop, index+48, 388
at index, 48
write →
1stop
```

The → in the -write- instruction is the arrow you created.

6. Condense your code. Watch the arrow move across the screen.
7. Return to your code. Replace your -doto- instruction with:

```
doto      1stop, index+40, 300, 2
```

8. Condense your code. Notice the "trail" left by your arrow. This is a result of having only one blank column in your character; you need two blank columns to avoid the trail.
9. Return to your character set block.
10. Create an arrow, leaving the *two* leftmost columns blank.
11. Return to the code containing your animation instructions.
12. Replace the character in your -write- instruction with your new character.
13. Condense your code. Your character no longer leaves a trail.
14. Experiment with animation. When you are comfortable with the process, continue with the next section of this activity (remain at the terminal).

LEAPING MOTION

You can make a character appear to move in "leaps" across the screen by using mode write, mode erase and a -doto- instruction. The following code causes a character to appear in four different positions, from your left to your right, with each position five coarse-grid spaces to the right of the last position:

```
define  index    = v1
        loc      = v2
calc    loc+3007  $$position where character
*                               first appears
unit    leap
doto    4jump, index+0, 3  $$index determines how many
*                               times character will
*                               appear and erase
at      loc+(index*5)  $$character position changes
*                               according to value of index
write   YOUR CHARACTER
catchup
pause   1              $$gives character the
*                               appearance of a pause
at      loc+(index*5)
mode    erase
write   YOUR CHARACTER  $$character is erased
```

```
mode      write      $$mode is changed so
*          writing will again be
*          visible
4j ump
```

The -catchup- instruction allows the terminal to finish displaying the character before the -pause- instruction is executed. You will learn more about the -catchup- instruction in activity 4-A.

Use the code for leaping motions to animate a character of your choice. You can create a new character, use a character you have already included in your code, or copy a character from the *aids* picture library. Experiment with the -at- instructions (change the multiple of *index*) to make larger jumps. Vary the value of *index* in the -doto- instruction to increase the number of jumps. When you finish experimenting, go on to your next assigned activity.

1-D. MORE AUTOMATED DISPLAYS



Identify steps required to use ID/SD to create circles and arcs; create broken circles and broken arcs; insert -arrow- instructions; create boxes, vectors, and broken lines; and insert other instructions not automatically inserted.

You've probably been thinking, "There's got to be an easier way to draw complicated diagrams with circles and arcs than the way I know." You're right. You learned to use automated displays to draw lines and write text in *PLATO Author Language: Part I*. This activity explains how to use automated displays to draw circles, arcs, broken circles, and broken arcs. It also explains how to create boxes, vectors and broken lines and how to insert -arrow- instructions while in ID/SD. In preparation for this exercise:

1. Sign on to a PLATO terminal as an author. Create a new block. Insert a -unit- instruction.
2. Enter ID mode. (Press *id1* and press NEXT to insert after your -unit- instruction.)

Complete each step of this section *exactly* as directed. If you make a mistake, start the section over.

AUTOMATED CIRCLES AND ARCS

This section of this exercise tells you how to create a circle or an arc, using ID/SD. The instructions for the circles and arcs you create will be automatically inserted in your code.

1. You should now be on the ID display. Move the cursor to the point at which you want the center of your circle to be.
2. Press the lowercase letter *o*.
3. Touch the screen at or move the cursor with the arrow keys to a point a short distance from the center point of your circle.
4. Press NEXT. Your circle is automatically drawn, and the instructions are automatically inserted in your code.
5. To create an arc, move the cursor away from the circle you created to the point at which you want your arc to start.
6. Press the uppercase letter *O*. You will be asked to choose arc (*a*) or circle (*c*).

7. Press the letter *a*. You will be asked to type a letter *y* (yes) if the arc is to be broken or a letter *n* (no) if the arc is not to be broken.
8. Press the letter *n*.
9. Touch the screen or move the cursor with the arrow keys to the end of the arc and press NEXT.
10. Move the cursor to a point through which the arc is to pass and press NEXT. Your arc is drawn and the instructions are automatically inserted in your code.
11. Move the cursor to the center point for your broken circle.
12. Press the uppercase letter *O*.
13. Press the letter *c* for circle; press the letter *y* for broken.
14. Move the cursor to a desired point on the outer edge of the broken circle and press NEXT. Your broken circle is drawn and the instructions are inserted in your code.
15. Create a broken arc in much the same way you created a continuous arc (steps 5 through 10); however, press the letter *y* for broken. Your broken arc will be automatically drawn and coded.
16. Press the SHIFT-BACK key. Notice that the instructions for your circles and arcs have been inserted in your code.


REMOVING OBJECTS IN ID/SD

This section of the exercise tells you how to remove an object from your code using ID/SD.

1. Use the ID/SD option to display the circle, the arc, the broken circle, and the broken arc. (Hint: Bring the appropriate lines of code to the top of the display. There should be eight lines of code. Type *sd8* and press NEXT. The four objects will be displayed; then you are in ID mode.)
2. To remove an object, the cursor must be at a point associated with it. To find the point associated with the broken circle, press LAB until the cursor is in the center of it.
3. Press the letter *r*. You will be asked whether you want to remove the object. Press the letter *y* for yes. The display will be replotted with the selected object removed.
4. Press SHIFT-BACK. The instructions that plotted the broken circle have been removed from your code.

THE AUTOMATED ARROW

This section shows you how to insert an -arrow- instruction using ID mode.

1. Enter ID mode. Position the cursor at the point where you want your arrow to be displayed.
2. Press the  key. An arrow will appear on the display and an -arrow- instruction will appear in your code.
3. Press the SHIFT-BACK key to return to your code.
4. Delete the code you have created in this activity.

ADDITIONAL AUTOMATED GRAPHICS OPTIONS IN ID/SD

In this section of the lesson, you will use ID/SD to create boxes, vectors, and broken lines automatically.

1. Create a new block and enter ID mode.
2. Move the cursor to one corner of a desired box and press the letter *b*.
3. Move the cursor to the opposite corner (diagonally) and press NEXT.
4. Type the number 3 to indicate a thickness of three lines. Press NEXT.
5. Your box with a thickness of three lines is drawn and coded. Experiment with this option. Use negative thickness such as -4. Notice that positive thickness are drawn outside the original box, while negative thickness are drawn inside.

Now you are going to experiment with drawing vectors (that is, lines with arrow heads).

6. Move the cursor to the desired location for the tail of a vector and press the lowercase letter *v*.
7. Move the cursor to the desired location for the arrowhead and press NEXT.
8. You will be asked to select the arrowhead size. Press NEXT for a regular size (approximately 11 dots).
9. Experiment with the *v* option in ID/SD. Try selecting a larger arrowhead by typing in a number larger than 11 (for example, try 25). Try creating a smaller arrowhead by typing in a number smaller than 10 (for example, 8) rather than using the default value of approximately 11.

In this next part, you will be drawing a broken line.

10. Move the cursor to the beginning of a desired broken line. Press the letter *p* to mark the point.
11. Move the cursor to another point on the display a short distance away from the first point. Press the uppercase letter *B*.

12. You will be asked for the dot length of the segments. The bigger your number, the longer your segments will be. Press the number 5; then press NEXT.
13. Draw another broken line; however, this time, select a dot segment length of 10.
14. Press SHIFT-BACK. The instructions have been inserted in your code.
15. Check the remaining space in your block. If you do not have at least 100 words remaining, delete some of the code or create a new block before continuing this exercise.

INSERTING INSTRUCTIONS

1. Insert a -unit- instruction. Enter ID mode.
2. Type in a question in text (*t*) mode. Press NEXT at the end of each line.
3. Press the BACK key *once*. The cursor appears.
4. Use the ⇐ key to create an -arrow- instruction.
5. Press the letter *i* to insert.
6. When you see the message *Line to insert*, type in an entire -answer- instruction (use the TAB key just as you would on your line display) and press NEXT.
7. Press the BACK key once to end insert mode.
8. Position the cursor where you want the feedback for your -answer- instruction to be.
9. Press *t* for text mode and type your feedback. Press NEXT after each line of feedback.
10. Press BACK once. Insert a -wrong- instruction in the same way you inserted the -answer- instruction.
11. Press BACK once. Position the cursor for the -wrong- feedback and type the feedback. Press NEXT after each line of feedback; then, press BACK once.
12. Insert an -endarrow- instruction. Press NEXT to enter it; press BACK to end insert mode.
13. Press SHIFT-BACK to return to your line display. Your entire unit is included in your code.

A few additional facts will make automated display creation easier:

1. When creating a complex display, return to the block display periodically to update your code; otherwise, a system failure can "wipe out" your display.
2. While you are editing in ID/SD, the amount of space remaining in your block is recorded in the lower right-hand corner of your display. If you exceed the maximum size of a block, you will not necessarily lose your code. Instead, you will be given the option to reduce the code, to ignore the code that will not fit, or to save the code so that you may insert it in another block.
3. Remember that if you press HELP while in ID/SD, a list of options appears on your screen. You need not learn these options at this time, but you may find it useful to investigate their uses at a later time.

CHECK YOUR UNDERSTANDING

Table 1 summarizes the ID/SD options you have learned. These options are sufficient for creating the title display called for in the programmable-ready material on the following pages. Study the programmable-ready material, and code it according to the instructions provided.

A transparency can be used to help you create the drawing. Trace the drawing with a marker. Then tape the transparency to your PLATO screen. Create lines and arcs that correspond to the lines and arcs on your transparency; you will find it much easier to space the drawing correctly if you use this method. Use the inserted transparency to help you create the drawing.

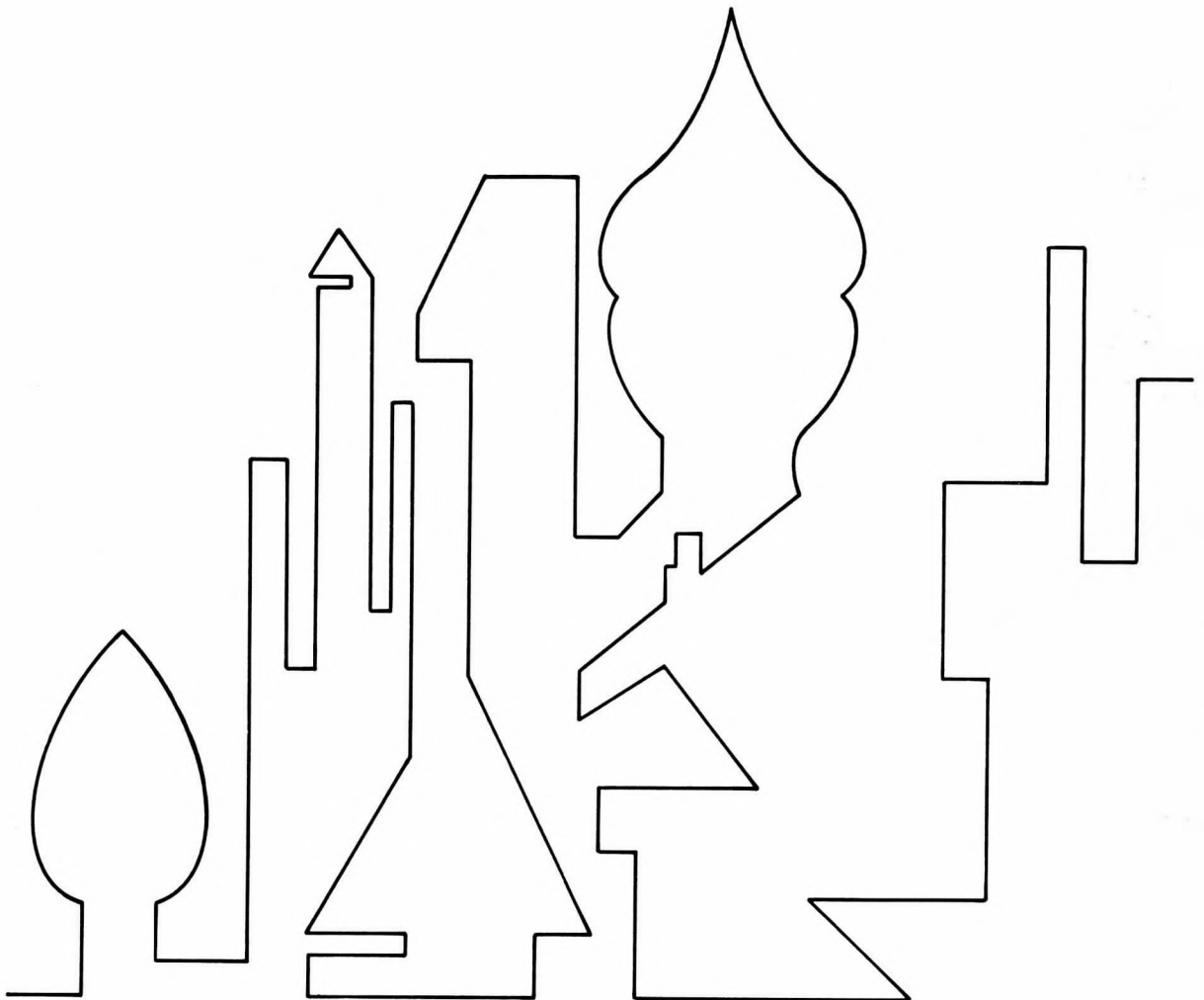
TABLE 1
ID/SD Directives

<i>Interactive Display Directive</i>	<i>Instruction Generated</i>	<i>Description or Effect</i>
f		For fine grid
g		For coarse (gross) grid
arrow keys (q, w, e, d, c, x, z, a)		To move the cursor (NOTE: the cursor may be moved by just touching the panel)
t	-write-	To insert text
	and	NEXT gives you another line
	-at-	BACK returns you to main display (NOTE: -at- is automatically inserted)
S	-size-	For -size- command
R	-rotate-	For -rotate- command
↔	-arrow-	For -arrow- command
o	-circle-	For -circle- command
O	-circle-	For 3-argument -circle- (arc)
O	-circleb-	For -circleb- command
O	-circleb-	For 3-argument -circleb- (arc)
p		To designate a beginning point
l	-draw-	To make a point to -draw- to
B	-draw-	To make a point to -draw- a broken line to
v	-vector-	To locate a vector on the display
b	-box-	To mark the corners of a box on the display

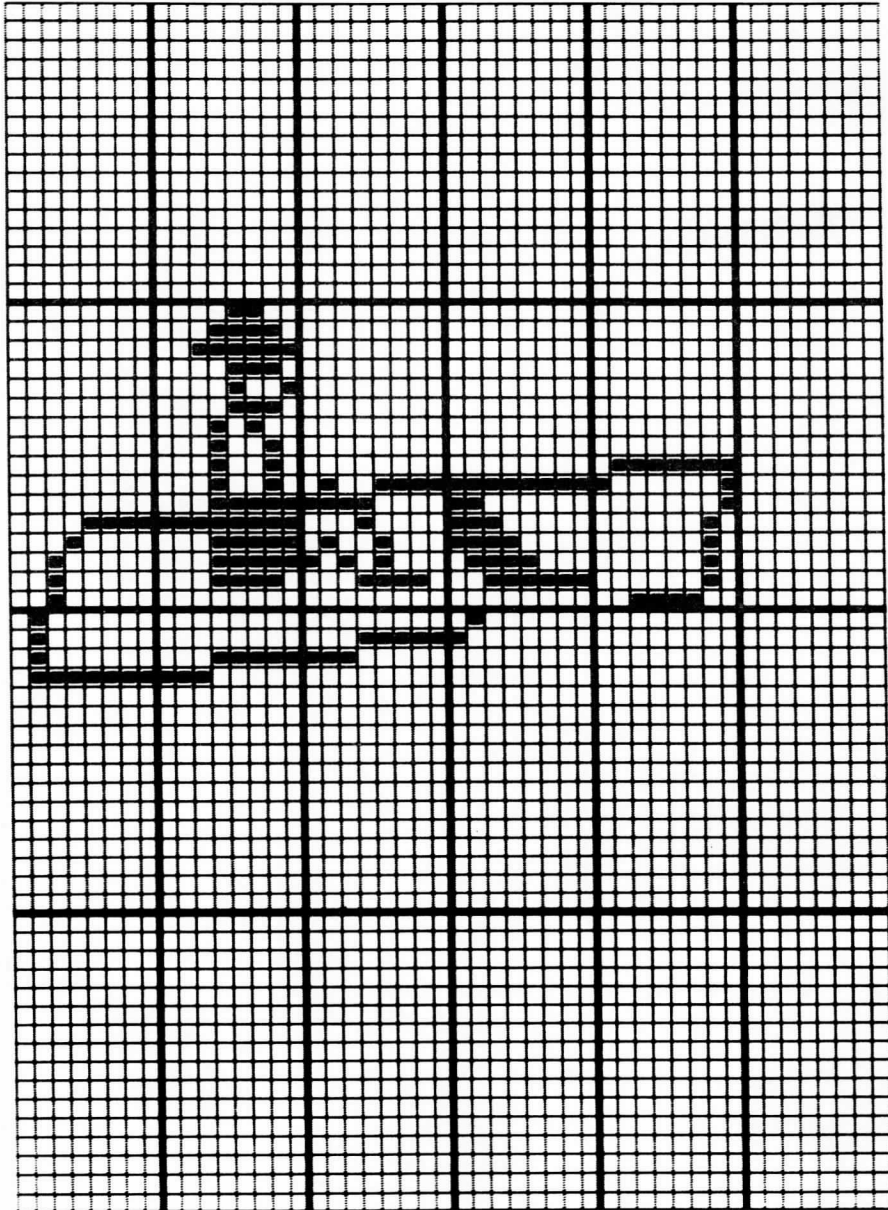
Directions: Create this title page for a lesson titled “Architecture of the Middle East.”

1. Center the title line; size it to approximate the size of the title as it appears on this page.
2. Create a flying carpet; animate it so it “flies” across the screen between the title and the drawing.

Architecture of the Middle East



If you do not have confidence in your artistic ability, you may wish to create your flying carpet based on the following printout.



2

Relocatable Graphics and Graphs

This unit explains how to create graphics and graphs that can be relocated by changing only one instruction. It also discusses how the `-size-` and `-rotate-` instructions affect relocatable graphics and presents instructions used to fill in relocatable graphs.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify the effects of the `-rorigin-`, `-rat-`, `-rdraw-`, `-rcircle-`, and `-rbox-` instructions
- Identify the effects that the `-size-` and `-rotate-` instructions have on relocatable graphics
- Identify the effects of the `-gorigin-`, `-axes-`, `-scalex-`, `-scaley-`, `-labelx-`, `-labely-`, `-markx-`, and `-marky-` instructions
- Identify the effects of the `-vbar-`, `-hbar-`, `-gbox-`, `-gcircle-`, and `-gdraw-` instructions

LEARNING ACTIVITIES

- _____ 2-A. Relocatable Graphics. Text/Exercise/CAI: This activity presents instructions that allow you to create relocatable graphics. Relocatable graphics can be repositioned by changing only one instruction.

- _____ 2-B. Sizing and Rotating Relocatable Graphics. Text/CAI: This activity shows you how the -size- and -rotate- instructions affect relocatable graphics.
- _____ 2-C. Creating Scaled Relocatable Graphs. Text/Exercise/CAI: This activity presents instructions used to create, scale, mark, and label the axes of relocatable graphs.
- _____ 2-D. Filling In Your Relocatable Graph. Text/Exercise/CAI: This activity presents instructions for filling in relocatable graphs with vertical and horizontal bars, boxes, circles, and lines.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of relocatable graphics and graphs, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

2-A. RELOCATABLE GRAPHICS



Identify the effects of the `-rorigin`-, `-rat`-, `-rdraw`-, `-rcircle`-, and `-rbox`- instructions.

Suppose you have drawn a picture using the `-draw`-, `-circle`-, and `-box`- commands. You want to use this picture a number of times at a number of different screen locations. With the instructions you now know, you would have to redraw the picture each time because of the coordinates specified in your instructions. Instructions of this type are in *absolute* form; they display at only one position.

This activity discusses *relocatable* graphics instructions. These instructions cause graphics to be displayed relative to a specified reference point. When the reference point changes, the position of the display changes. Relocatable instructions discussed in this activity are `-rorigin`-, `-rat`-, `-rdraw`-, `-rcircle`-, and `-rbox`-.

THE `-rorigin`- INSTRUCTION

The reference point for relocatable instructions is specified in the `-rorigin`- instruction. The `-rorigin`- instruction can take one of three forms:

```
rorigin FINE GRID COORDINATES  
rorigin VARIABLE OR EXPRESSION X, VARIABLE OR EXPRESSION Y  
rorigin
```

The first format specifies a reference point using fine-grid coordinates. Fine grid should always be used; coarse grid sometimes results in unanticipated displays. An example of this format is:

```
rorigin 256,256
```

This instruction establishes a reference point at the middle of the screen.

The second format allows variables and expressions to be used to establish a reference point if the position of the display is dependent on action that has taken place previously in the lesson.

The third format, `-rorigin`- with a blank tag, sets the origin at the correct *wherex* and *wherey* (the fine-grid equivalents of the system variable *where*).

The setting of an -rorigin- remains in effect until another -rorigin- is executed. If no -rorigin- is specified, it is automatically set at 0,0.

THE -rat- INSTRUCTION

The -rat- instruction is similar to the -at- instruction in that it establishes a left-hand margin for the display of text, data, or figures. However, this position is relative to the preceding -rorigin-. For example,

```
rorigin 256,256
rat      0,0
write    Where will
         this write?
```

produces this display:

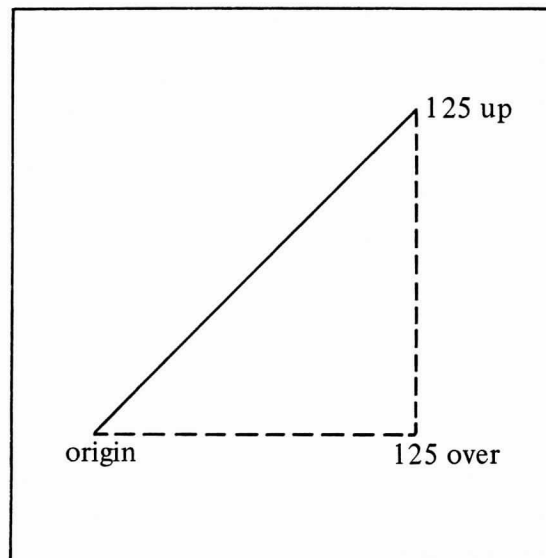


THE -rdraw- INSTRUCTION

The -rdraw- instruction is similar to the -draw- instruction; lines are drawn between specified coordinates. The position of the display, however, is relative to the position set in the -rorigin- instruction. For example:

```
rorigin 256,256
rdraw   0,0;125,125
```

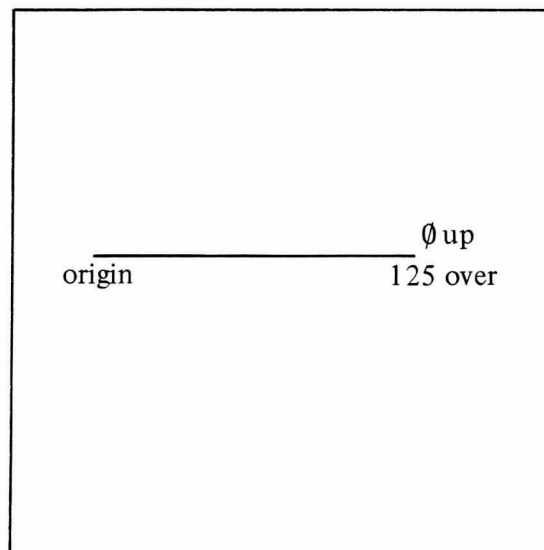
draws a line from the origin to a point 125 dots over from the origin and 125 dots up:



This code:

```
rorigin 256,256
rdraw 0,0;125,0
```

draws a line from the origin to a point 125 dots over from the origin and 0 dots up:



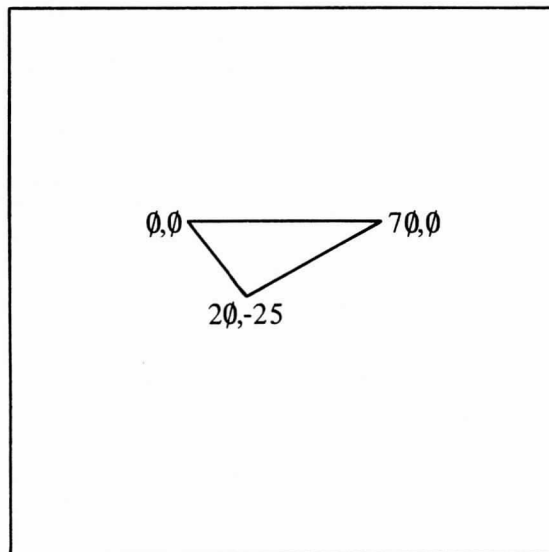
THE -rdraw- INSTRUCTION IN A -doto- LOOP

The -rdraw- instruction can be put into a -doto- loop. This section of this activity will be taught via the PLATO terminal. Follow these steps:

1. Sign on to the PLATO system as an author.
2. Enter this code in an empty block:

```
define loop      = v1
unit    rdraw
rorigin 256,256
doto    1fini,loop+0,330,30
rotate  loop
rdraw   0,0;70,0;20,-25;0,0
1fini
```

This code establishes your reference point at 256,256. Starting at 0, the variable *loop* will be incremented by 30 until 330 is reached. The variable *loop* is the number of degrees to rotate; it is repeated twelve times. The -rdraw- instruction draws a line 70 dots over and 0 dots up and a line 20 dots over and 25 dots down (because of the -25) from the origin. It then draws a line from this point back to the -rorigin-:



3. Condense your code to see the results.
4. Return to your code.
5. Now you will move this drawing to another position. To do so, code another unit with a new unit name, and a new -rorigin- instruction:

```
rorigin 350,256
```

Use the same instructions for your -doto- loop as you used in step 2 of this exercise (the *s* and *is* directives can help you here). Do *not* save and insert your -define- instruction; you only need to define the variable once.

6. Condense your code and look at the drawing in the first unit. Press NEXT to see the drawing in the second unit; you have moved the drawing by changing only the -rorigin- instruction.
7. Remain at the terminal for the next section of this activity.

If you want to use a drawing throughout an activity, changing only its location on the display, you can save code by coding all relocatable instructions in an auxiliary unit, specifying a different -rorigin- instruction each time the unit is done:

```
unit      five
rorigin 256,384
do        figure
***
unit      figure $$auxiliary unit
doto     1fini,loop=0,330,30
rotate   loop
rdraw    0,0;70,0;20,-25;0,0
1fini
```

THE -rcircle- INSTRUCTION

The -rcircle- instruction allows you to draw a relocatable circle or arc. The format is much the same as that of the -circle- instruction:

```
rcircle RADIUS
rcircle RADIUS,ANGLE 1,ANGLE 2 $$for an arc
```

You can set the center of your circle with the -rorigin- instruction or include a -rat- instruction to set it elsewhere. To see how the -rcircle- instruction works, complete these steps:

1. Establish your first reference point:

```
unit      draw
rorigin 100,100
do        circle
```

2. Code this auxiliary unit:

```
unit    circle
rcircle 50
rcircle 25
rcircle 6
rcircle 3
```

3. Establish a second origin in another unit *before* your auxiliary unit, using the same set of circles in a different position:

```
unit    redraw
rorigin 300,300
do      circle
```

4. Condense your code. Notice that your set of circles appears at a different position on the second display than it did on the first display.
5. Practice using the `-rcircle-` instruction to create relocatable circles and arcs.
6. Remain at the terminal.

THE `-rbox-` INSTRUCTION

The `-rbox-` instruction is similar to the `-box-` instruction in that if two corners and an optional thickness are set in the tag, a box will be drawn. The position of the box, however, is relative to the point set in the `-rorigin-`. Three formats of the `-rbox-` instruction are:

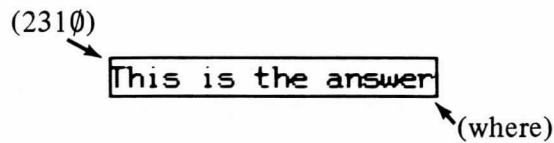
```
rbox    CORNER1;CORNER2  (FINE GRID OR COARSE GRID)
rbox    CORNER1;CORNER2;THICKNESS (FINE GRID OR COARSE GRID)
rbox    ;CORNER2  (FINE GRID OR COARSE GRID)
rbox    CORNER2  (FINE GRID OR COARSE GRID)
```

The first draws a box with its corners relative to the `-rorigin-`. The second allows a thickness to be specified.

The third is a special case. It automatically sets one corner at *where*; the second corner is specified. This is useful when boxing text. For example, if the words *This is the answer* begin at relative position 0,0, this instruction

```
rbox    ;0,16
```

produces these results:



The fourth format draws a box with one corner at the origin, and the other at the specified distance from the origin.

USING ID/SD TO CREATE RELOCATABLE GRAPHICS

When you understand the principles of relocatable graphics instructions, you can go on to code these instructions easily and automatically using ID/SD directives. Follow these steps:

1. Enter ID mode.
2. Move the cursor to the point at which you want your origin to be.
3. Press SHIFT-DATA. This sets your origin and automatically puts the -rorigin- instruction in your code. Other drawings made on this display will be in relocatable form.
4. To draw a line from the origin to another point, move the cursor to the other point and press the letter *l* for line.
5. Continue to move the cursor and to press the letter *l* until you have the drawing you desire.
6. To begin a new drawing that is not connected to the previous drawing, move the cursor to another point on the display. Press the letter *p* to mark the location.
7. Continue to move the cursor and to press the letter *l* to complete the drawing. Draw other shapes using the ID directives *o*, *O*, *v*, *b*, and *B*.
8. Press the SHIFT-BACK key. Your -rdraw- instruction and other relocatable instructions have been automatically inserted in your code.
9. Practice using ID/SD to insert relocatable instructions in your code.
10. When finished, sign off the terminal and go on to the next activity.

SUMMARY

The following chart summarizes the information presented on relocatable graphics instructions.

<i>Command</i>	<i>Tag</i>	<i>Effect</i>
rorigin	fine-grid coordinates	Specifies a reference point with fine-grid coordinates
	variablex,variabley	Specifies a reference point using variables or expressions
	blank	Sets origin at current <i>wherex,wherey</i>
rdraw	x, y;x,y	Draws relocatable lines from the position specified in the first argument to the position specified in the second argument
rcircle	radius	Draws a relocatable circle
rbox	corner1;corner2	Draws a relocatable box with corners at specified points
	corner1;corner2;thickness	Draws a relocatable box of a specified thickness with corners at specified points
	;corner2	Draws a relocatable box with one corner at <i>wherex,wherey</i> and the other at a specified point
	corner2	Draws a relocatable box with one corner at the origin and the other at a specified point

Relocatable graphics can be created in ID/SD by establishing an origin with SHIFT-DATA.

2-B. SIZING AND ROTATING RELOCATABLE GRAPHICS



Identify the effects that the `-size-` and `-rotate-` instructions have on relocatable graphics.

Relocatable instructions allow you to reposition graphics; you can also size and rotate your relocatable graphics. Because of the `-rorigin-` and `-rat-` instructions, however, you must take care when inserting `-size-` and `-rotate-` instructions in your code. This activity allows you to experiment using the `-size-` and `-rotate-` instructions with relocatable instructions.

THE `-size-` INSTRUCTION

The `-size-` instruction can be used to make relocatable graphics larger or smaller. Remember, however, that all relocatable instructions are relative to the point set in your `-rorigin-` instruction. The `-size-` instruction, if placed before a `-rat-` instruction, sizes the `-rat-` instruction. For example, in this code:

```
rorigin 50,50
size    2
rat     100,200
```

the arguments of the `-rat-` instruction are doubled; the `-rat-` instruction now sets a position 200 dots over and 400 dots up from the point set by the `-rorigin-`. You must take care that a `-size-` instruction does not run your drawing over the edge of the screen.

THE `-rotate-` INSTRUCTION

The `-rotate-` instruction also affects any relocatable instructions it precedes, including the `-rat-` instruction. Thus, you must take care that the placement of your `-rotate-` instructions does not move your graphics off the screen.

Figure 1 illustrates how the `-rotate-` instruction pivots a relocatable figure. The figure is always positioned by the `-rat-` instruction at a certain point relative to the origin. When rotated, it remains the same distance from the origin, but pivots according to the degrees specified. If pivoted so any portion runs over the screen,

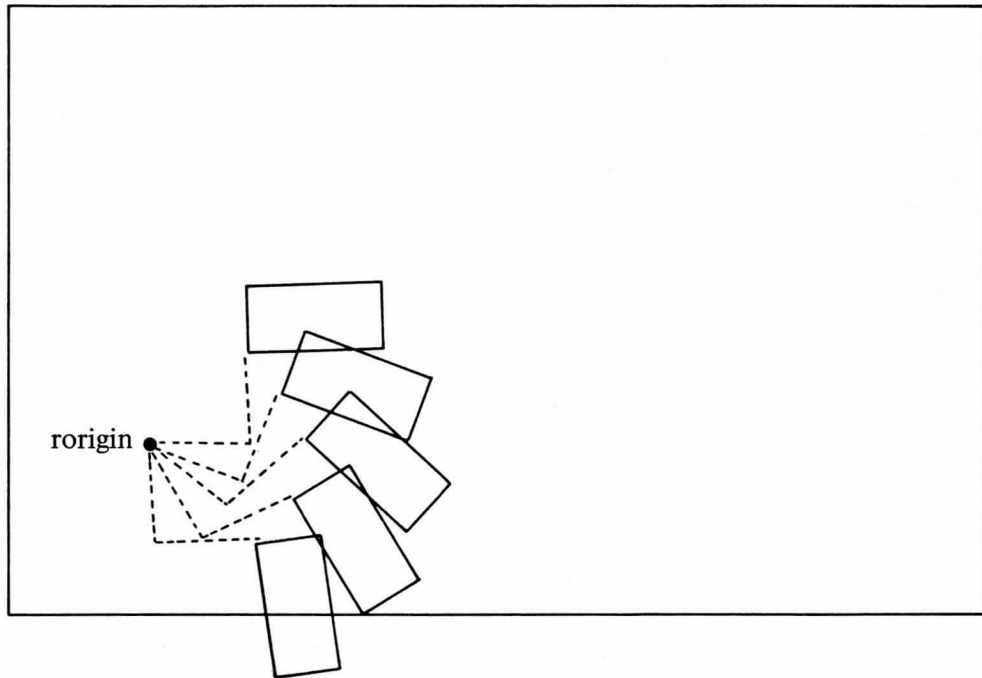


Figure 1. The -rotate- instruction pivots a relocatable figure

the wraparound of the lines creates strange displays. The bottom square on the diagram would cause this figure to wrap around; the portion of the rectangle shown below the line in the diagram would actually shoot out in strange directions from other sides of the screen.

There is an instruction that can be used to eliminate the wraparound effect. The -window- instruction specifies an area on the display beyond which no line drawings will be displayed:

```
window  LOWER LEFT COARSE GRID; UPPER RIGHT COARSE GRID
window  LOWER LEFT FINEX, FINEY; UPPER RIGHT FINEX, FINEY
window                                     $$removes the effect
```

Thus a -window- 0,0; 511,511 will set the display limits at the lower-left and upper-right corners of the display; no wraparound effect will occur.

CAI INTRODUCTION

It is easier to determine how the -size- and -rotate- instructions affect a figure when you can see the results. This short CAI lesson allows you to experiment with a figure by changing the origin, size and degrees of rotation.

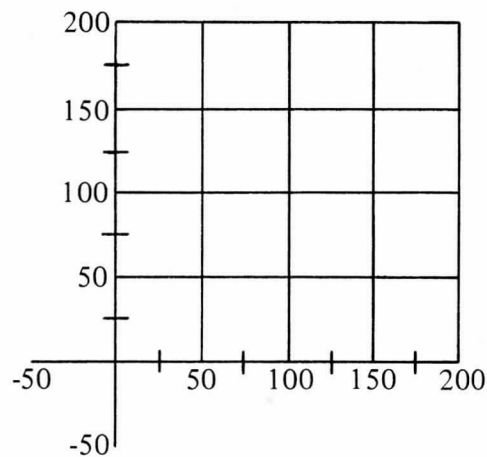
Now sign on to a Control Data PLATO terminal. When you see the list of modules, type the letter of this unit. Then press NEXT. A module description display will appear. Press the letter *a* to see your assignments. When you locate the description of this CAI activity, "Effects of Relocatable Graphics Instructions," press LAB to take the lesson.

2-C. CREATING SCALED RELOCATABLE GRAPHS



Identify the effects of the `-gorigin-`, `-axes-`, `-scalex-`, `-scaley-`, `-labelx-`, `-labeley-`, `-markx-`, and `-marky-` instructions.

You can draw this graph using the instructions you learned in *PLATO Author Language: Part I*.



It would be time consuming, and could not be relocated easily, however. In this activity, you will learn to draw a graph using a series of graphing instructions that make the creation of graphs much easier. The following instructions are discussed: `-gorigin-`, `-axes-`, `-scalex-`, `-scaley-`, `-labelx-`, `-labeley-`, `markx-`, and `-marky-`. You will be given an opportunity to practice using these instructions.

THE `-gorigin-` INSTRUCTION

The `-gorigin-` instruction specifies the reference point for the other graphing instructions. For example,

```
gorigin 128,112
```

sets the origin at a position 128 dots over and 112 dots up from the lower left-hand corner of the screen.

The -gorigin- instruction can also use a coarse-grid tag:

```
gorigin 1624
```

If no -gorigin- instruction is included, the origin is automatically set at 0,0.

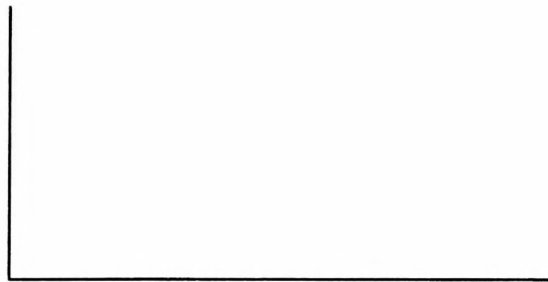
THE -axes- INSTRUCTION

The -axes- instruction specifies the length in dots of the x and y axes of the graph and draws these axes; x is the horizontal axis and y is the vertical axis.

The arguments of the -axes- instruction are specified in fine grid. They can be either positive or negative. For example,

```
axes      200,100
```

creates this display. The x axis is drawn from the origin to a point 200 dots over. The y axis is drawn from the origin to a point 100 dots up.



If the instruction is:

```
axes      -50,-25,200,100
```

this display results:



A \emptyset cannot be included in a two-argument tag of an -axes- instruction. If you don't want a line to appear, use the number 1 for an argument.

A blank -axes- instruction tag will draw the axes according to the tag of the last -axes- instruction. Thus, if the axes erase when a unit is completed and you want them to reappear in the next unit, use the -axes- instruction with a blank tag.

THE -scalex- AND -scaley- INSTRUCTIONS

The -scalex- and -scaley- instructions establish the value of the rightmost point on the x axis and the highest point on the y axis without marking or labeling these points. For example,

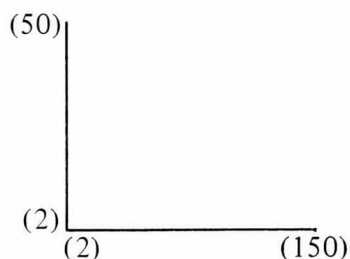
```
scalex  150
scaley  300
```

assigns the rightmost point of the x axis the value of 150, and the highest point of the y axis the value of 300. Other instructions following the -scalex- and -scaley- instructions will be scaled accordingly, as you will see in subsequent sections of this activity.

An argument can be included in the -scalex- or -scaley- instruction that establishes the coordinate of the intersection of the x and y axes at a number other than \emptyset . For example,

```
axes      100, 100
scalex    150, 2
scaley    50, 2
```

establishes 150 as the rightmost point, and 2 as the point of intersection for the x axis; 50 is established as the highest point, and 2 as the point of intersection for the y axis:



The numbers shown are *not* produced by these instructions, only the scale.

The -scalex- and -scaley- instruction should be included directly after the -axes- instruction so all following graphing instructions will be scaled correctly. If no

-scalex- or -scaley- instruction is included, the arguments of the -axes- instruction determine the scale. The -axes- instruction

```
axes      100,200
```

sets the rightmost point of the x axis at 100, and the highest point of the y axis at 200.

THE -labelx- AND -labeLy- INSTRUCTIONS

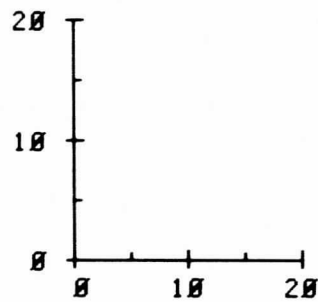
The -labelx- and -labeLy- instructions mark and label the x and y axes. Their formats are:

```
labelx    MAJOR MARKS
labeLy    MAJOR MARKS
labelx    MAJOR MARKS,MINOR MARKS
labeLy    MAJOR MARKS,MINOR MARKS
labelx    MAJOR MARKS,MINOR MARKS,MARK LENGTH
labeLy    MAJOR MARKS,MINOR MARKS,MARK LENGTH
```

The first argument marks and labels the axes with major or larger marks at specified intervals. The second argument marks, but does not label, the axes with minor or smaller marks at the specified intervals. For example, these instructions

```
gorigin 256,256
axes     100,100
scalex   20
scaley   20
labelx   10,5
labeLy   10,5
```

produce the following graph. The rightmost and highest points of the axes are labeled 20 according to the scales established by the -scalex- and -scaley- instructions. The first argument in the -labelx- and -labeLy- instructions causes the larger, labeled marks to appear at intervals of ten. The second argument of the -labelx- and -labeLy- instructions causes smaller, unlabeled marks to appear at intervals of five. The marks and numbers on a graph change according to the numbers specified in the arguments of the -labelx- and -labeLy- instructions.



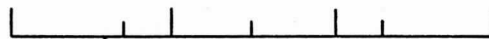
The minor marks argument must be a factor of the major marks argument. These tags are acceptable:

```
label x 50,10
label x 20,5
label y 240,80
label y 60,20
```

These are not:

```
label x 60,25
label x 240,100
label y 80,6
label y 100,30
```

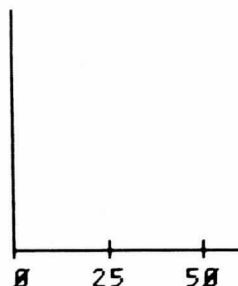
The PLATO system does *not* allow you to mark an axis like this:



The major marks argument, however, need not be a factor of the -scalex- or -scaley- instructions. This code:

```
gorigin 256,256
axes 100,100
scalex 60
label x 25
```

produces this axis:



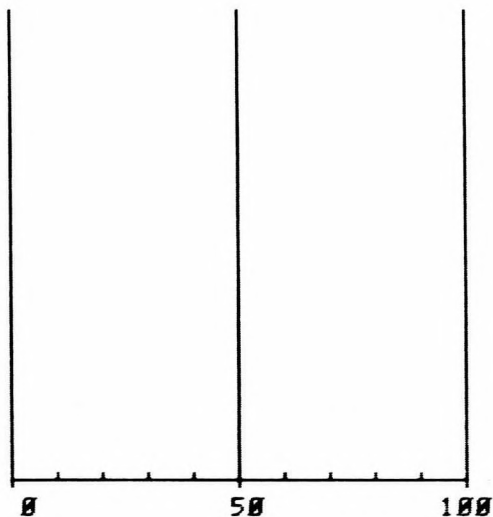
Axes can also be labeled with decimals, such as:

```
labelx 2.5
labely 3.5
```

The third argument of the `-labelx-` and `-labely-` instructions specifies the mark length. If the mark length argument is `0` or absent, normal marks, such as in all previous examples, are drawn. If the mark length is `1`, major marks are extended to the boundaries of the graph. These instructions

```
gorigin 100,100
axes 200,200
scalex 100
labelx 50,10,1
```

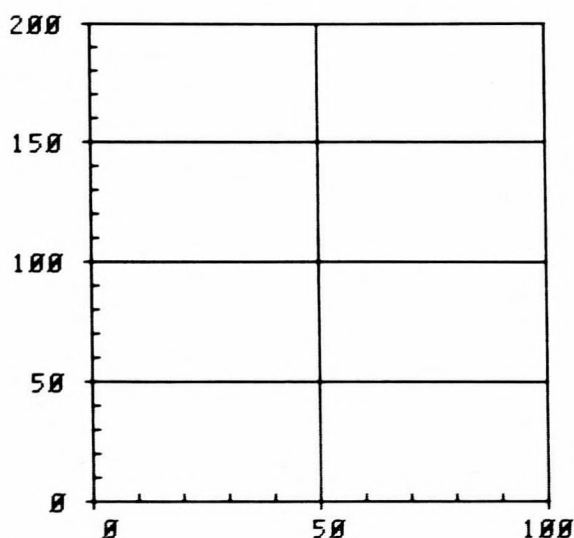
produce this graph:



These instructions

```
gorigin 100,200
axes 200,200
scalex 100
labelx 50,10,1
labely 50,10,1
```

produce this graph:



Notice that the y axis is scaled according to the y argument in the -axes- instruction because no -scaley- instruction has been included. This code

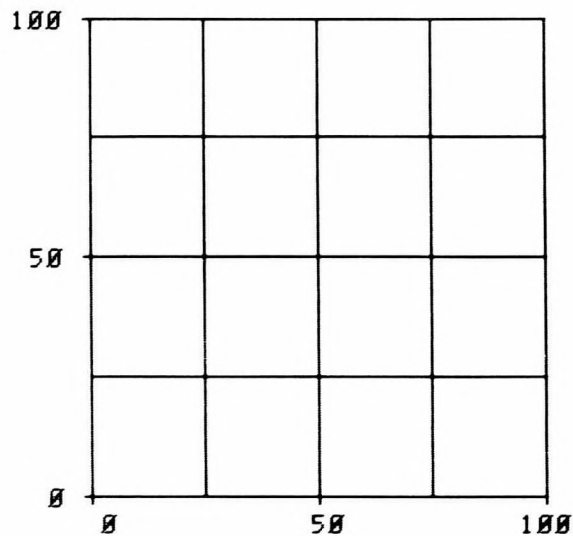
```
gorigin 100,200
axes    200,200
scalex  100
scaley  100
labelx  50,10,1
labely  50,10,1
```

makes the marks of both the x and y axes the same.

A mark length of 2 extends both major and minor marks to the boundaries of the graph. These instructions

```
gorigin 100,200
axes    200,200
scalex  100
scaley  100
labelx  50,25,2
labely  50,25,2
```

produce this graph:



THE -markx- AND -marky- INSTRUCTIONS

The -markx- and -marky- instructions use the same general format as the -labelx- and -labele- instructions:

```
markx  MAJOR MARKS, MINOR MARKS, MARK LENGTH
marky  MAJOR MARKS, MINOR MARKS, MARK LENGTH
```

The minor marks and mark length arguments are optional.

The -markx- and -marky- instructions simply mark the axes; they do not label them.

If your programming will involve logarithmic scales, request the tutor feature *logarithm scales* in lesson aids for appropriate graphing instructions.

CHECK YOUR UNDERSTANDING

In this section of this activity, you will use the instructions presented thus far to draw and label the axes of graphs.

1. Sign on to a PLATO terminal as an author.
2. Enter an empty block.

3. Type in these instructions:

```
unit    graph
gorigin 250,250
```

This is the reference point for all of your other graphing instructions.

4. Add the instruction:

```
axes    100,200
```

5. Condense your code. Your x axis runs 100 dots over from your origin. Your y axis runs 200 dots up from your origin.
6. Return to your code.
7. Replace your -axes- instruction with:

```
axes    -20,-20,100,200
```

8. Condense your code. Notice that your x and y axes begin 20 dots before the origin.
9. Return to your code.
10. Delete all lines of code in *unit graph*.
11. Now you will create an aesthetically pleasing graph and mark the axes. Insert these instructions:

```
unit    graph
gorigin 100,100
axes    240,240
scalex  60
scaley  80
labelx  10
labely  10
```

Notice that the -axes- instruction is evenly divisible by the arguments of the other graphing instructions.

12. Condense your code. Notice that the x axis is divided into six sections (the scaled length of 60 is divided into segments at intervals of 10), and that the y axis is divided into eight sections (the scaled length of 80 is divided into segments at intervals of 10). Major marks have been displayed and labeled because only one argument is included in the -labelx- and -labely- instructions.
13. Return to your code. Replace the -scaley- instruction with:

```
scaley  80,20
```

14. Condense your code. Notice that the point where the y axis intersects the x axis is now labeled 20. Note also that even though the -labeledy- instruction remains the same, there are only six major marks; they are still at intervals of 10, but because the initial point is now 20 rather than 0, two intervals have been deleted.

15. Return to your code. Replace the -labeledx- instruction with:

```
labeledx 25
```

16. Condense your code. Notice that the PLATO system places major marks at intervals of 25, but that no mark is made at the end of the x axis; a -scalex- instruction of 60 permits marks only to 60. If the -scalex- instruction tag is not evenly divisible by the -labeledx- instruction tag, labeling is performed as you see on the display.

17. Return to your code. Replace the -scaley- instruction with:

```
scaley 60
```

18. Condense your code. Notice that the last mark on your y axis is now 60.

19. Return to your code. Replace your -labeledx- and -labeledy- instructions with:

```
labeledx 10
```

```
labeledy 10
```

20. Condense your code. Note that your axes are marked and labeled at intervals of 10.

21. Return to your code. You will now see how the minor marks argument affects your graphing displays. Replace your -labeledx- and -labeledy- instructions with:

```
labeledx 10,5
```

```
labeledy 10,5
```

22. Condense your code. Note that minor marks have been placed at intervals of five along your x and y axes.

23. Return to your line display. Replace your -labeledx- instruction with:

```
labeledx 10,5,1
```

24. Condense your code. Notice that the major marks on your x axis extend to the boundaries of your graph because of the mark length argument in your -labeledx- instruction.

25. Return to your code. Replace your -labeledy- instruction with:

```
labeledy 10,5,1
```

26. Condense your code. The major marks on your y axis now extend to the boundaries of your graph.
27. Return to your code. Replace your -labelx- and -labeley- instructions with:

```
labelx 10,5,2  
labeley 10,5,2
```

28. Condense your code. Note that a mark length argument of 2 extends all marks to the boundaries of your graph.
29. Return to your code. Change your -gorigin- instruction to:

```
gorigin 200,200
```

30. Condense your code. Your graph has been relocated.
31. Return to your code. Replace your -gorigin- instruction with:

```
gorigin 300,300
```

32. Condense your code. Notice that the -gorigin- instruction causes strange results; it set a position too close to the edge of the screen.
33. Return to your code. Replace the -gorigin- instruction with:

```
gorigin 100,100
```

Delete your -scalex- and -scaley- instructions. Note the arguments in your -axes- instruction.

34. Condense your code. Note that your -axes- instruction now scales your graph. The -labelx- and -labeley- instructions still mark at intervals of 10 and 5, and extend all marks, but there is no longer room to label all major marks; the PLATO system determines which marks will be labeled.
35. Return to your code. Replace your -labelx- and -labeley- instructions with:

```
labelx 40  
labeley 40
```

36. Condense your code. The PLATO system can now mark and label all major marks.
37. Return to your code. Replace your -labelx- and -labeley- instructions with:

```
labelx 30  
labeley 30
```

38. Condense your code. Note that marks are made at intervals of 30, but that major *and* minor marks appear. If all were major marks and were labeled, they would be packed too closely together. Thus, the PLATO system reacts as though the -labelx- and -labeLy- instructions were:

```
labelx 60,30  
labeLy 60,30
```

It won't let you make a cluttered graph.

39. Return to your code. Replace your -labelx- and -labeLy- instructions with:

```
markx 10  
marky 10
```

40. Condense your code; notice that the axes are marked, but not labeled.
41. Experiment with the instructions presented in this activity until you are comfortable with their use.
42. Sign off the terminal.

SUMMARY

The following chart summarizes the information presented on relocatable graphing instructions.

<i>Command</i>	<i>Tag</i>	<i>Effect</i>
gorigin	fine grid coordinates coarse grid coordinates	Establishes the reference point for other graphing instructions
axes	length of x axis, length of y axis blank	Specifies the length of the x and y axes Draws the axes according to the tag of the previous -axes- instruction
scalex	number for last point on x axis	Establishes the value of the rightmost point on the x axis. Sets scale for other graphing instructions
scaley	number for last point on y axis	Establishes the highest point on the y axis. Sets scale for other graphing instructions
labelx	major marks major marks,minor marks major marks,minor marks,mark length	Marks and labels the x axis
labeley	major marks major marks,minor marks major marks,minor marks,mark length	Marks and labels the y axis
markx	major marks major marks,minor marks major marks,minor marks,mark length	Marks, but does not label, the x axis
marky	major marks major marks,minor marks major marks,minor marks,mark length	Marks, but does not label, the y axis

2-D. FILLING IN YOUR RELOCATABLE GRAPH



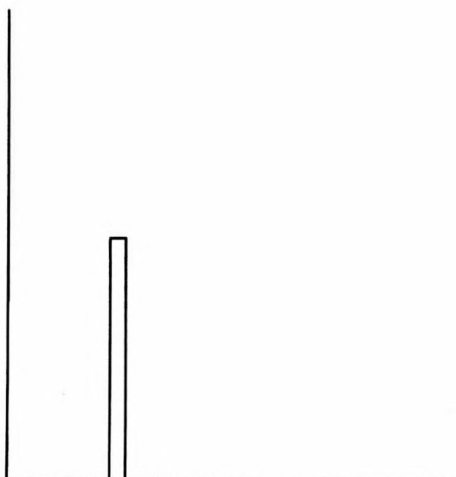
Identify the effects of the `-vbar-`, `-hbar-`, `-gbox-`, `-gcircle-`, and `-gdraw-` instructions.

When you have drawn, scaled, and labeled the axes of your relocatable graph, you can begin to “fill in” the graph with meaningful data. This activity explains how to:

- Draw vertical bars with the `-vbar-` instruction
- Draw horizontal bars with the `-hbar-` instruction
- Draw boxes with the `-gbox-` instruction
- Draw circles and lines with the `-gcircle-` and `-gdraw-` instructions

VERTICAL AND HORIZONTAL BARS

To create a relocatable bar graph, use the `-vbar-` and `-hbar-` instructions to draw bars from the appropriate axes to a specified point. The `-vbar-` instruction draws a vertical bar:



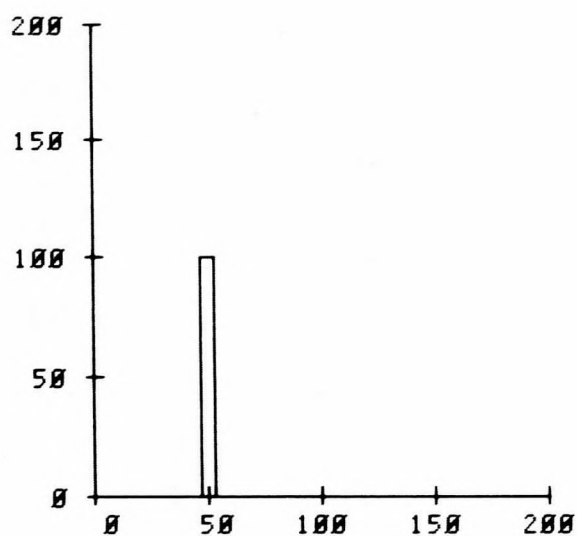
Its basic format is:

`vbar` `X POSITION FOR CENTER OF BAR, Y POSITION FOR HEIGHT`

These instructions

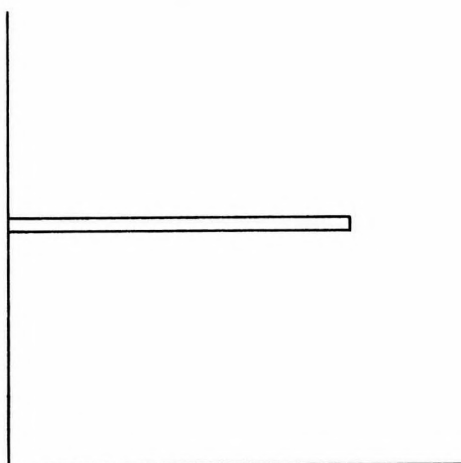
```
gorigin 256,256  
axes 200,200  
labelx 50  
labely 50  
vbar 50,100
```

create this display:



The x and y positions specified in the -vbar- tag are relative to the origin and the scaled axes.

The -hbar- instruction draws a horizontal bar:



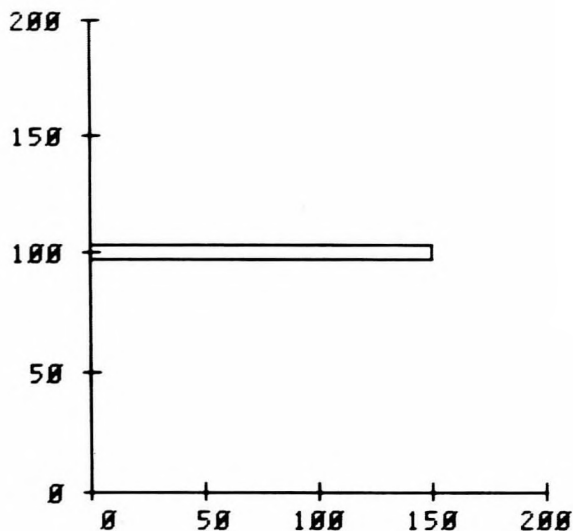
Its basic format is:

`hbar` X POSITION FOR END OF BAR, Y POSITION FOR CENTER

These instructions

```
gorigin 256,256
axes    200,200
labelx  50
labely  50
hbar    150,100
```

create this display:

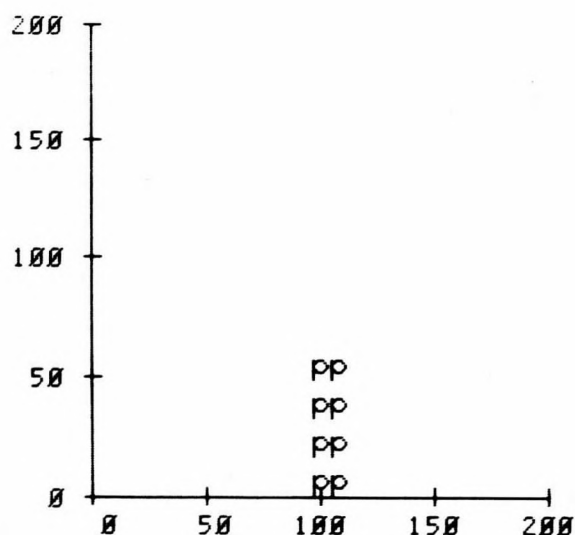


The horizontal bar is also relative to the origin and the scaled axes.

The `-vbar-` and `-hbar-` instructions can draw bars using character strings. These instructions

```
gorigin 256,256
axes    200,200
labelx  50
labely  50
vbar    100,50,pp
```

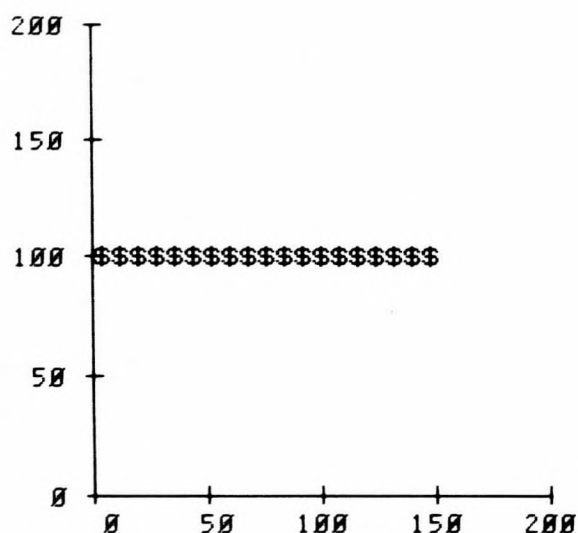
create this display:



These instructions

```
gorigin 256,256
axes    200,200
labelx  50
labely  50
hbar    150,100,$
```

create this display:



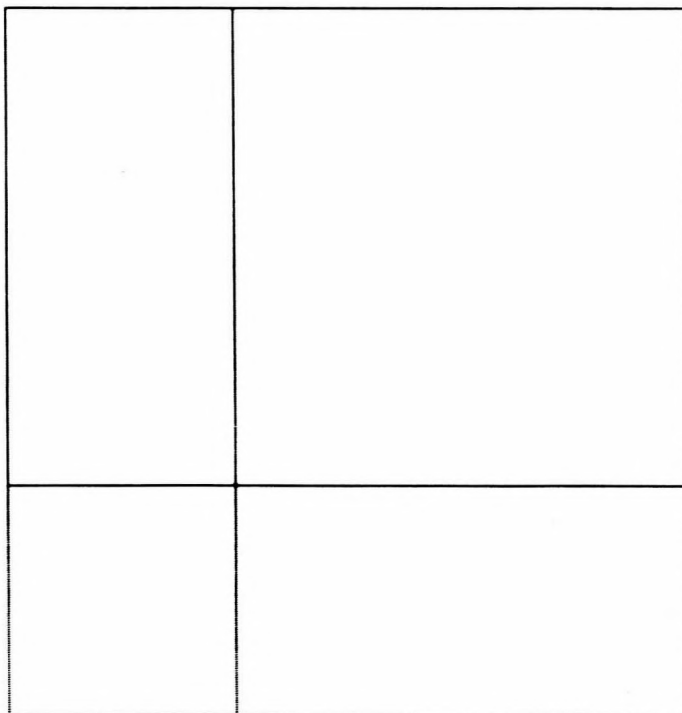
Any character string can be used with the exception of two \$ signs. (Two \$ signs indicate a comment in code.)

BOXES

The -gbox- instruction with a blank tag draws a box around the boundaries set by the previous -axes- instruction. These instructions

```
gorigin 200,200  
axes    -100,-100,200,200  
gbox
```

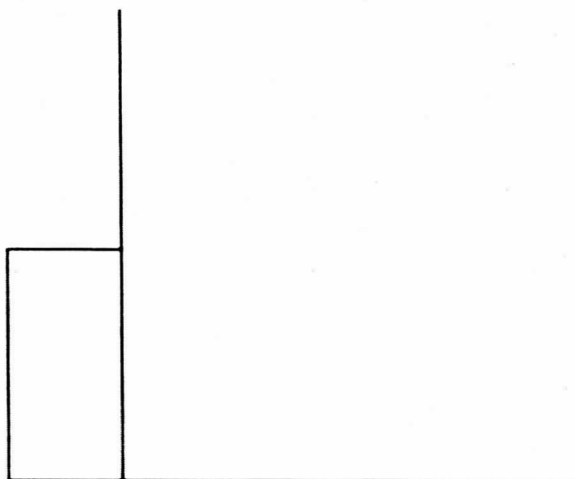
create this display:



A -gbox- instruction with a two-argument tag draws a box with one corner at the origin and the other corner at the x and y axes locations specified. These instructions

```
gorigin 100,100  
axes    200,200  
gbox    -50,100
```

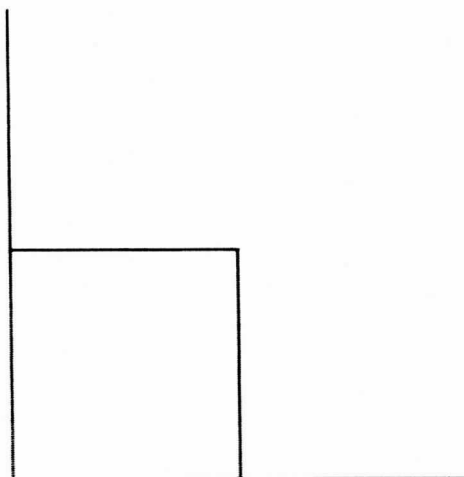
create this display:



These instructions

```
gorigin 100,100
axes    200,200
gbox    100,100
```

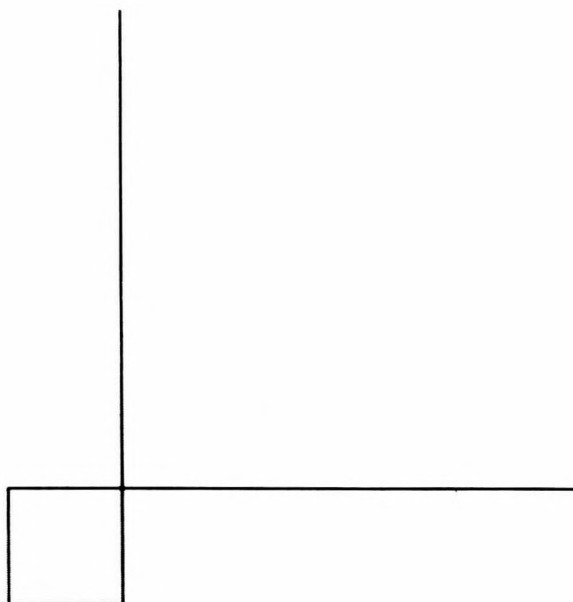
create this display:



These instructions

```
gorigin 100,100
axes    200,200
gbox    -50,-50
```

create this display:



The box is scaled according to the scale set by the `-axes-` instruction or the `-scalex-` and `-scaley-` instructions.

The four-argument `-gbox-` instruction is similar to the `-box-` instruction. Two opposite corners are specified for the box. A thickness argument is optional. The format is:

```
gbox      XCORNER1,YCORNER1;XCORNER2,YCORNER2;THICKNESS
```

The box is drawn relative to the origin, and scaled by the `-axes-` or `-scalex-` and `-scaley-` instructions.

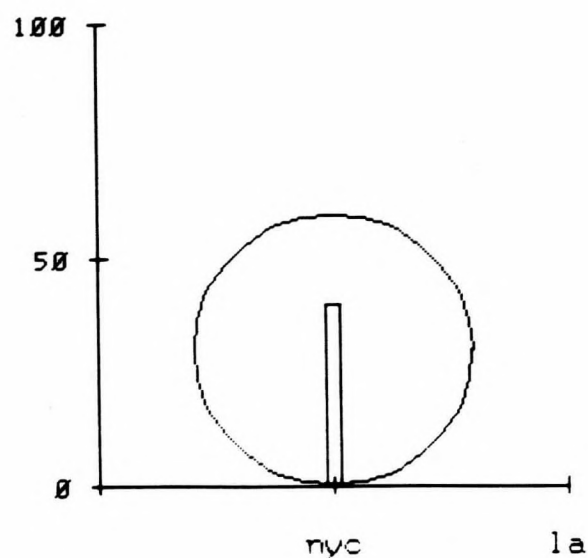
CIRCLES

To draw a relocatable circle within your graph, use the `-gat-` instruction and the `-gcircle-` instruction. These instructions have the same purposes and general formats as the `-at-` and `-circle-` instructions, but are relative to the origin and the scaled axes.

For example, these instructions

```
unit      g1
gorigin   50,50
axes      200,200
scalex    100
scaley    100
vbar      50,40
markx     50
labeley   50
gat       50,-16
at        wherex-12,wherey
write     nyc
gat       100,-16
at        wherex-8,wherey
write     la
gat       50,30
gcircle   30
```

produce this graph:



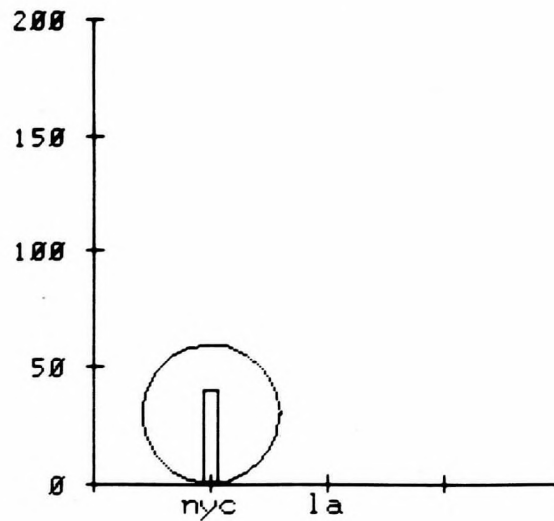
If the scale is changed, the size of the circle changes. The same instructions with no -scalex- or -scaley- instructions

```

unit      g2
gorigin  50,50
axes      200,200
vbar      50,40
markx     50
labeledy  50
gat       50,-16
at        wherex-12,wherey
write     nyc
gat       100,-16
at        wherex-8,wherey
write     la
gat       50,30
gocircle  30

```

produce this graph:



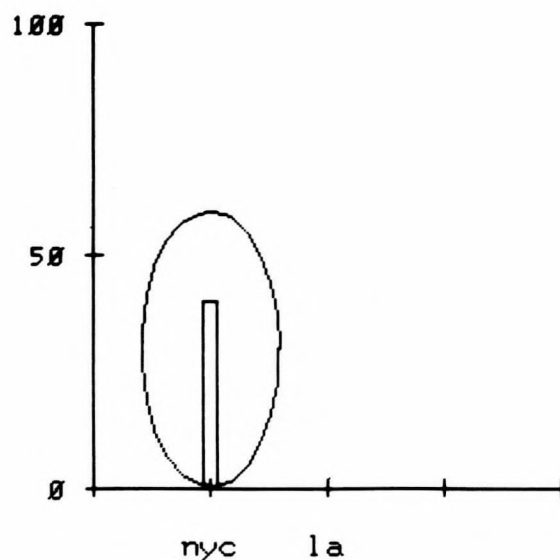
If the scales of the x and y axes differ (if the `-scalex-` and `-scaley-` instructions are different, or if the x and y axes of the `-axes-` instruction are different in the case of no `-scalex-` and `-scaley-`), you can create an oval with the `-gcircle-` instruction. These instructions

```

unit      g3
gorigin   50,50
axes      200,200
scaley    100
vbar      50,40
markx     50
labely    50
gat       50,-16
at        wherex-12,wherey
write     nyc
gat       100,-16
at        wherex-8,wherey
write     la
gat       50,30
gcircle   30

```

produce this graph:



Notice the use of the `-at-` instruction in the last three examples. Because the letters are not scaled, they must be centered. Thus, their position is set with a `-gat-` instruction so they can be relocated. They are centered, however, with an `-at-` instruction that makes use of the system variables `wherex` and `wherey`, and the width of the characters in dots.

The center of the letters *nyc* is first set with the instruction:

gat 50, -16

Each letter is eight dots wide; together, the three letters are 24 dots wide. Because the letter *y* should be directly below the 50 mark on the *x* axis, the letters must be moved to the left. The instruction

at wherex-12,wherey

moves the entire character string 12 dots to the left. Thus, 12 dots of the 24 dot character string are to the left of the 50 mark and 12 dots are to the right of the 50 mark. The character string is centered. The *wherey* used in the *-at-* instruction tag causes the character string to remain at the *y* location specified in the *-gat-* instruction.

LINES

To draw lines within your graph, use the *-gdraw-* instruction; it has the same purpose and general format as the *-draw-* instruction, but is relative to the origin and the scaled axes.

CHECK YOUR UNDERSTANDING

Directions: Sign on to a PLATO terminal as an author. Use the instructions for creating and filling in a graph to:

1. Draw vertical bars on a graph with the *-vbar-* instruction.
2. Draw horizontal bars on a graph with the *-hbar-* instruction.
3. Draw vertical and horizontal bars of character strings using the *-vbar-* and *-hbar-* instructions.
4. Draw a box around a graph using the *-gbox-* instruction with a blank tag.
5. Draw a box outside the graph using the *-gbox-* instruction.
6. Draw a box inside the graph using the *-gbox-* instruction.
7. Draw a perfect circle around a vertical bar with the *-gcircle-* instruction.
8. Draw an oval around a vertical bar by making your *-scaley-* instruction a lower number than your *-scalex-* instruction or *-axes-* instruction *x* argument.
9. Center some letters below a mark on a graph.

SUMMARY

The following chart summarizes the information presented on instructions used to fill in relocatable graphs.

<i>Command</i>	<i>Tags</i>	<i>Effect</i>
vbar	x position for bar center, y position for bar height, optional character string	Draws a relocatable vertical bar
hbar	x position for end of bar, y position for center of bar, optional character string	Draws a relocatable horizontal bar
gbox	blank	Draws a box around the boundaries set by the previous -axes- instruc- tion
	x,y	Draws a box with one cor- ner at the origin and the other at a specified point
gcircle	radius	Draws a relocatable circle
gat	location .	Positions circles and text
gdraw	from point;to point	Draws a line from the first specified point to the sec- ond specified point

3

Variable Manipulation

This unit contains a variety of methods for making the best use of your variable storage space, including segmentation and arrays. It also discusses additional variables not previously discussed in either PLATO Author Language course. These include common and storage variables, datasets, and namesets. Ways to manipulate variables, including functions and argument passing, are also discussed.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Convert binary and octal numbers to decimal numbers, octal numbers to binary numbers, and binary numbers to octal numbers
- Recognize ways in which information is stored in floating point and integer variables
- Identify correct ways to segment variables and reference segmented variables
- Identify the effects of the reassignment of values and the use of system-defined variables on storage space
- Recognize steps in the processes of defining one-, two-, and three-dimensional arrays
- Identify correct uses of system-defined and author-defined functions
- Identify characteristics of common and storage variables
- Identify characteristics of namesets and datasets
- Identify correct ways to pass arguments

LEARNING ACTIVITIES

- _____ 3-A. Numbering Systems. Text/Exercise: This activity explains the basic principles of numbering systems. It discusses and provides practice in converting binary and octal numbers to decimal numbers, octal numbers to binary numbers, and binary numbers to octal numbers.
- _____ 3-B. Variable Formats. Text/Exercise: This activity explains the types of information stored in floating point and integer variables and discusses how the information is stored.
- _____ 3-C. Segmented Variables. Text/Exercise: The process of segmenting variables to enable variables to hold more than one value is explained.
- _____ 3-D. Stretching Your Variables. Text: This activity discusses how values in variables can be reassigned. It also presents commonly used system-defined variables.
- _____ 3-E. Author-Defined Arrays. Text/Exercise: This activity discusses the processes of defining one-, two-, and three-dimensional arrays.
- _____ 3-F. System-Defined and Author-Defined Functions. Text/Exercise: This activity explains how to use system- and author-defined functions to make your programming more efficient.
- _____ 3-G. Common and Storage Variables. Text/Exercise: This activity discusses two types of variables, common and storage, that can be used to increase the storage space of a lesson.
- _____ 3-H. Datasets and Namesets. Text: This activity discusses general characteristics of datasets and namesets. These files can be used to increase storage space.
- _____ 3-I. Argument Passing: Another Way to Assign Values. Text: This activity explains how -do-, -join-, and -jump- instructions can be used to assign values to variables.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of variable manipulation, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.



3-A. NUMBERING SYSTEMS

Convert binary and octal numbers to decimal numbers, octal numbers to binary numbers, and binary numbers to octal numbers.

You probably use the decimal numbering system for everyday calculations, but the PLATO system does not. It uses the binary and octal numbering systems to calculate and store. Before you use the data manipulation instructions presented in this unit, you should understand the principles of these numbering systems. You should also be able to convert binary and octal numbers to decimal numbers, octal numbers to binary numbers, and binary numbers to octal numbers. The importance of these skills will be made more clear in subsequent activities, when your conversion skills are used.

NUMBERING SYSTEMS

Numbering systems differ in the number of symbols or numerals they have. For example, the decimal system has 10 numerals: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. To write a number higher than 9, combinations of numerals are used. Ten, for example, needs two numerals, 1 and 0.

The word numeral stands for a numerical symbol; 1 is a numeral as are 4 and 0. A numeral differs from a number in that the numeral is a symbol itself, while a number stands for some quantity; 9 is a numeral that can also be a number standing for nine things. However, 12 is not a numeral; it is a number made up of two numerals. In this activity, the word *numeral* means one of the symbols in a numbering system. When the word *number* is used, it stands for a quantity.

THE BINARY SYSTEM

A numbering system is named for the number of numerals it has. The binary system has only two numerals; 0 and 1. For a quantity higher than one, numerals such as 2, 5, or 7 cannot be used because they do not exist in the binary system. Thus, to indicate the quantity *two* in the binary system, you need two places; *two* is written as 10. To show that this is a binary number, you can write a subscript 2 beside the binary number, like this: 10_2 .

Table 2 indicates the binary numerals used to represent the decimal quantities 1 through 16.

TABLE 2
Decimal Numbers and Corresponding Binary Numbers

<i>Quantity (In Decimal)</i>	<i>Binary Numbers</i>
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

THE OCTAL SYSTEM

The octal system has eight symbols or numerals, 0 through 7. The numeral 3 is used to write the quantity three in octal. To show that it is octal, you write an 8 subscript: 3₈. To write the number seven in octal, you would write 7₈.

Suppose you have a quantity of more than seven, such as eight. No numeral 8 exists in the octal system, so you have to use two of the numerals that you do have: 1 and 0. This looks like decimal ten, so the subscript 8 should be added: 10₈. The quantity nine in octal is 11₈. In the octal system, two numerals are used in the first and second places to show quantities up through 77₈ (equal to 63 in decimal).

For higher numbers, you need three places. You start at 100_8 (equal to 64 in decimal), and proceed until you reach 777_8 (equal to 511 in decimal). For even higher numbers, use four or more places.

The subscript of any number indicates the *base* of the number; it shows the number of symbols or numerals in that numbering system. If the system has ten numerals, its base is 10. The octal system has a base of 8. The binary system has a base of 2.

CONVERTING BINARY NUMBERS TO DECIMAL NUMBERS

To convert a binary number to decimal, you must find the value of each numeral and total the values. The values of the positions increase by a multiple of 2 with each place added to the left. The first (rightmost) place value is always 1. For example, with 10_2 , the 0 is in the 1s position and has no value. 2 is the value in the 2s place. So 10_2 in binary equals 2 in decimal.

The number 11_2 is converted as follows: The first numeral has a value of 1. The second numeral has a value of 2. The sum of these values is 3. Thus, 11_2 in binary equals 3 in decimal.

Now consider a larger number, 1011_2 . The numbers 1, 2, 4, and 8 represent the place values from right to left. Thus, the value of the first place is 1 (the binary numeral times the place value). The value of the second numeral is 2 (1×2). The value of the third numeral is 0 (0×4). The value of the fourth numeral is 8 (1×8). The sum of these values ($1 + 2 + 0 + 8$) is 11; thus, 1011_2 in binary equals 11 in decimal.

The values of the first eight places in the binary system are:

512	128	64	32	16	8	4	2	1
				$(2 \times 2 \times 2 \times 2)$	$(2 \times 2 \times 2)$	(2×2)	(1×2)	

To convert any number in the binary system to decimal, multiply the numeral in each position by the place value of its position, and total the values.

CHECK YOUR UNDERSTANDING**Part 1**

Directions: In the blank to the right of each combination of binary numerals, write the decimal equivalent. The numbers at the top of the columns are the place values for the columns.

	8	4	2	1	
1.	0	1	0	0	_____
2.	0	1	1	0	_____
3.	1	0	1	0	_____
4.	1	1	0	0	_____
5.	1	1	0	1	_____
6.	0	1	1	1	_____
7.	0	0	1	0	_____
8.	1	0	0	1	_____
9.	1	1	1	1	_____
10.	1	0	1	1	_____

Answers

- | | |
|-------|--------|
| 1. 4 | 6. 7 |
| 2. 6 | 7. 2 |
| 3. 10 | 8. 9 |
| 4. 12 | 9. 15 |
| 5. 13 | 10. 11 |

Part 2

Directions: Convert the following binary numbers into decimal numbers. The place values are 1, 2, 4, 8, 16, 32, 64, 128, 256, from right to left.

1. 11000001 _____
2. 110011 _____
3. 101010 _____
4. 1011101 _____
5. 10010010 _____
6. 111001110 _____
7. 1111000000 _____
8. 110011100 _____
9. 10101010 _____
10. 111111100 _____

Answers

- | | |
|--------|---------|
| 1. 97 | 6. 462 |
| 2. 51 | 7. 480 |
| 3. 42 | 8. 412 |
| 4. 93 | 9. 170 |
| 5. 146 | 10. 508 |

CONVERTING OCTAL NUMBERS TO DECIMAL

This section discusses how numbers are converted from octal to decimal. The octal system works the same way as the binary system. For example, the values for the first seven places in base 8 are shown below:

262144	32768	4096	512	64	8	1
			(8x8x8x8)	(8x8x8)	(8x8)	

If the octal number is 10, the value for the first place is 0. The value for the second place is 8. Thus, octal 10 is equal to 8+0 or 8 in decimal.

If the octal number is 100, the 1 in the third place has a value of 64. Therefore 100₈ is equal to 64 in the decimal system.

If the octal number is $1\emptyset11$, the fourth place has a value of 512. Thus, $1\emptyset11_8$ is equal to $512 + \emptyset + 8 + 1$ or 521 in decimal.

The numbers $1\emptyset\emptyset$ and $1\emptyset11$ look like binary numbers because only the numerals 1 and \emptyset are used. The octal system, however, has numerals from \emptyset to 7, and these numerals can be used in any of the places. If the number is $7\emptyset\emptyset$ in octal, the value of the third place is 7×64 or 448, making $7\emptyset\emptyset_8$ equal to 448 in decimal.

A CONVERSION EXERCISE

This section steps you through the process of converting octal numbers to decimal numbers.

1. To convert 1473_8 to decimal, multiply the numeral in each place by the value for that place. (The number is 1473_8 .)

The value for the first numeral is _____.

The value for the second numeral is _____.

The value for the third numeral is _____.

The value for the fourth numeral is _____.

(3, 56, 256, 512)

2. Now add the values of all the places. The sum of those values is the decimal equivalent of the octal number. The sum is _____.

(827)

3. Now convert 236_8 to decimal.

Multiply the numeral in each place by the place value:

The value for the first numeral is _____.

The value for the second numeral is _____.

The value for the third numeral is _____.

(6, 24, 128)

4. Find the sum of the values. What is the sum? _____

(158)

This sum is the decimal equivalent.

CONVERTING OCTAL NUMBERS TO DECIMAL USING A TABLE

Table 3 can help you quickly find the decimal value for octal numerals in places 0 through 7. For example, if the octal numeral 1 is in the third place from the right, the decimal value is 64. If the octal numeral 4 is in the third place, its decimal value is 256. If the octal numeral 7 is in the fifth place, its value is 28672. Having found the values for the numerals, you need only add the values to determine the decimal equivalent to an octal number.

TABLE 3
Octal to Decimal Conversion Chart

<i>Octal Numeral</i>	<i>Decimal Values</i>									
0	0	0	0	0	0	0	0	0	0	0
1	16,777,216	2,097,152	262,144	32,768	4,096	512	64	8	1	
2	33,554,432	4,194,304	524,288	65,536	8,192	1,024	128	16	2	
3	50,331,648	6,291,456	786,432	98,304	12,288	1,536	192	24	3	
4	67,108,864	8,388,608	1,048,576	131,072	16,384	2,048	256	32	4	
5	83,886,080	10,485,760	1,310,720	163,840	20,480	2,560	320	40	5	
6	100,663,296	12,582,912	1,572,864	196,608	24,576	3,072	384	48	6	
7	117,440,512	14,680,064	1,835,008	229,376	28,672	3,584	448	56	7	

The following columns show places and numerals. Using the Octal to Decimal Conversion Chart, fill in the blanks below with the correct decimal values.

	<i>Numeral</i>	<i>Place</i>	<i>Decimal Value</i>
1.	5	2	_____
2.	7	2	_____
3.	3	3	_____
4.	6	3	_____
5.	4	4	_____
6.	5	4	_____
7.	7	5	_____
8.	6	5	_____
9.	3	6	_____
10.	5	7	_____

Answers

- | | |
|---------|-------------|
| 1. 40 | 6. 2560 |
| 2. 56 | 7. 28672 |
| 3. 192 | 8. 24576 |
| 4. 384 | 9. 98304 |
| 5. 2048 | 10. 1310720 |

CHECK YOUR UNDERSTANDING

Directions: Convert these numbers to decimal. Use the chart if you prefer.

<i>Octal</i>	<i>Decimal Value</i>
1. 67	_____
2. 245	_____
3. 624	_____
4. 1637	_____
5. 5610	_____
6. 20414	_____
7. 424555	_____
8. 372117	_____
9. 1420000	_____
10. 3004772	_____

Answers

- | | |
|---------|------------|
| 1. 55 | 6. 8460 |
| 2. 165 | 7. 141677 |
| 3. 404 | 8. 128079 |
| 4. 927 | 9. 401408 |
| 5. 2952 | 10. 788986 |

CONVERTING BINARY TO OCTAL

To convert binary numbers to octal numbers, you could first change them to decimal, and then to octal. There is, however, an easier way. Remember this rule: *Three binary digits equal one octal digit*. Divide the binary number into sections of three digits each. Find the decimal value of each group of three. Then string (don't add) the numbers together. To convert 1101001110110_2 to octal, you would divide the binary number and assign values to the divisions of the number as shown:

$$1/101/001/110/110$$

$$1 \quad 5 \quad 1 \quad 6 \quad 6$$

Incomplete groups are counted as a value. The octal number 15166 is equivalent to the binary number 1101001110110 .

CHECK YOUR UNDERSTANDING

Directions: Convert these binary numbers to octal.

<i>Binary</i>	<i>Octal</i>
1. 11	_____
2. 110	_____
3. 1111	_____
4. 1010	_____
5. 10110	_____
6. 111000	_____
7. 111010	_____
8. 1010101	_____
9. 10110111001	_____
10. 11100100010101	_____

Answers

1. 3

2. 6

3. 17 $\begin{array}{c} 1 \quad 7 \\ \hline (1111) \end{array}$ 4. 12 $\begin{array}{c} 1 \quad 2 \\ \hline (1010) \end{array}$ 5. 26 $\begin{array}{c} 2 \quad 6 \\ \hline (10110) \end{array}$ 6. 70 $\begin{array}{c} 7 \quad 0 \\ \hline (111000) \end{array}$ 7. 72 $\begin{array}{c} 7 \quad 2 \\ \hline (111010) \end{array}$ 8. 125 $\begin{array}{c} 1 \quad 2 \quad 5 \\ \hline (1010101) \end{array}$ 9. 2671 $\begin{array}{c} 2 \quad 6 \quad 7 \quad 1 \\ \hline (10110111001) \end{array}$ 10. 34425 $\begin{array}{c} 3 \quad 4 \quad 4 \quad 2 \quad 5 \\ \hline (11100100010101) \end{array}$ **CONVERTING OCTAL NUMBERS TO BINARY NUMBERS**

To convert octal numbers to binary numbers, remember the same rule: *Three binary digits equal one octal digit*. Each octal digit in your octal number must be represented by three binary digits. For example, the number 574₈ could be converted to binary in this way:

$$\begin{array}{ccc} 5 & 7 & 4 \\ 101 & 111 & 100 \end{array}$$

The binary number for 574₈ is 101111100.

CHECK YOUR UNDERSTANDING

Directions: Convert these octal numbers to binary numbers.

<i>Octal</i>	<i>Binary</i>
1. 4	_____
2. 25	_____
3. 374	_____
4. 2056	_____
5. 3721	_____
6. 54327	_____
7. 75642	_____
8. 462157	_____
9. 3611175	_____
10. 20657711	_____

Answers

$$1. \begin{array}{c} 4 \\ \hline 100 \end{array}$$

$$2. \begin{array}{cc} 2 & 5 \\ \hline 10101 \end{array}$$

$$3. \begin{array}{ccc} 3 & 7 & 4 \\ \hline 11111100 \end{array}$$

$$4. \begin{array}{cccc} 2 & 0 & 5 & 6 \\ \hline 10000101110 \end{array}$$

$$5. \begin{array}{cccc} 3 & 7 & 2 & 1 \\ \hline 11111010001 \end{array}$$

$$6. \begin{array}{ccccc} 5 & 4 & 3 & 2 & 7 \\ \hline 1011000110111 \end{array}$$

$$7. \begin{array}{ccccc} 7 & 5 & 6 & 4 & 2 \\ \hline 111101110100010 \end{array}$$

$$8. \begin{array}{cccccc} 4 & 6 & 2 & 1 & 5 & 7 \\ \hline 10011001000110111 \end{array}$$

$$9. \begin{array}{cccccc} 3 & 6 & 1 & 1 & 1 & 7 & 5 \\ \hline 11110001001001111101 \end{array}$$

$$10. \begin{array}{cccccc} 2 & 0 & 6 & 5 & 7 & 7 & 1 & 1 \\ \hline 1000011010111111001001 \end{array}$$

SUMMARY

To convert binary or octal numbers to decimal:

1. Multiply each numeral in each position by the place value of that position.
2. Total the values.

To convert binary numbers to octal numbers:

1. Divide the binary number into sections of three digits each.
2. Convert each section to a decimal number (incomplete groups also count as a decimal number).
3. String (don't add) the numbers together.

To convert octal numbers to binary numbers:

1. Convert each octal numeral into a three-digit binary number.
2. String (don't add) the binary numbers together.



3-B. VARIABLE FORMATS

Recognize ways in which information is stored in floating point and integer variables.

To this point, you have used this format when defining and using variables: v1,v2,v3....v15. There is another format that can be used, depending on what you want to store and retrieve: n1,n2,n3.....n15.

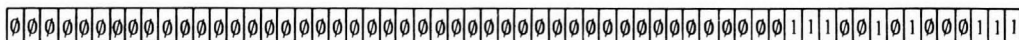
The variables νl and $n l$ store information in the same place in the computer's memory; thus, if combinations of n and ν variables are used, you still have only 150 variable storage spaces.

What is the difference in these two types of variables besides their format? They store information in a different way; this is important because some pieces of information must be stored one way and some the other. This activity explains:

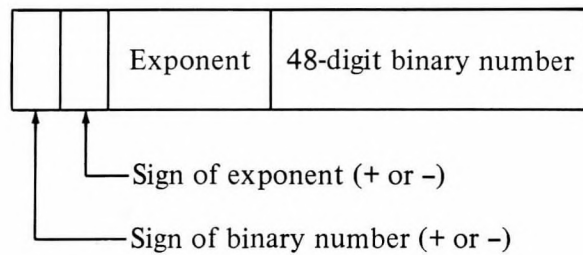
- How information is stored in v (floating point) and n (integer) variables
- What types of information are stored in each type of variable

STORAGE OF v VARIABLES

Each variable, whether in *v* format or *n* format, can hold sixty *binary digits*, or *bits*. This may lead you to expect that your *v* variables store numbers in this way:



But they *don't*. The ν variables allow for the storage of decimal fractions and very large numbers. Because the decimal point position of a number stored in a ν variable floats based on an exponent, the ν type variable is called a floating point variable. Storage of information in a ν variable is similar to scientific notation. Scientific notation uses exponents and coefficients to store decimal numbers in powers of ten. 93,000,000 is 9.3×10^7 in scientific notation. The ν variable storage form uses a similar type of notation to use exponents to store *binary* numbers. The exponent is what tells the computer where to store the decimal point. The sixty bits of the ν variable are used in this way:



One bit is reserved for the sign of the binary number. One bit is reserved for the sign of the exponent. Ten bits are reserved for the exponent, and forty-eight bits are reserved for the binary number. These spaces are reserved whether they are needed or not.

You don't have to completely understand the storage format of floating point variables, but a general knowledge of how storage is accomplished should show you why decimals are always stored in *v* variables, and why, when you are in doubt as to how something will be stored, you should use *v* variables for *-define-* and *-calc-* instructions.

STORAGE OF *n* VARIABLES

The *n* variables can be used to store whole numbers or *integers*. Thus, they are often called *integer* variables. Integer variables store information in the way you might expect it to be stored. The number is converted to binary digits; the sign of the number is stored in the leftmost bit, and one binary digit is stored in each of the remaining fifty-nine spaces.

Integer variables can also store numeric representations of characters, such as letters of the alphabet. Because both numbers and character strings are stored as binary numbers, you must indicate to the computer when you wish to store and retrieve characters (letters, numerals, and some symbols) rather than values. To do so, use either double or single quotation marks around the character string to be stored. Single quotes left-justify the stored characters in the variable; they tell the computer to store the characters on the left side of the variable. Double quotes right-justify the stored characters. Thus, if you want to store the character string *had* on the left side of variable *n1*, you would use this instruction:

```
calc      n1 ← 'had'
```

To show a character string, use the *-showa-* instruction. For the previous example, the instruction

```
showa     n1
```

shows the letters *had*.

If you want to store numerals, and show them in their original form, use an *n* variable, quotes, and the -showa- instruction, as in this example:

```
calc      n2 ← '002'
showa     n2
```

Otherwise, the computer will store the value (2 in this case) rather than the numerals (002).

DETERMINING CHARACTER STORAGE SPACE REQUIREMENTS

Six binary bits (two octal digits) are reserved for each character you store in a variable. The shift itself counts as a character when uppercase characters are being stored. Thus, you can store up to ten lowercase characters ($10 \times 6 = 60$ bits) or five uppercase characters in each variable. Character strings longer than ten character codes require additional variables.

Each character on the keyboard has been assigned an octal *character code* or *internal code*. The codes are assigned in octal because octal is easier to read than binary, and it can be converted to binary easier than decimal. To see the internal code for any character, request the TUTOR feature *internal codes* from the *aids* lesson.

When converted to binary, these internal codes will show you the binary numbers the computer uses to store your character string. Table 4 shows a few of the character codes.

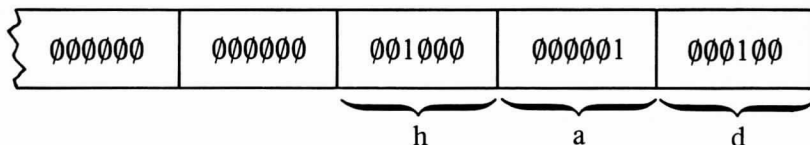
TABLE 4
Character Codes

<i>Character</i>	<i>Internal Code</i>	<i>Binary Equivalent</i>
a	o01	000 001
b	o02	000 010
c	o03	000 011
d	o04	000 100
e	o05	000 101
f	o06	000 110
g	o07	000 111
h	o10	001 000
	o55	101 101
A	o7001	111 000 000 001

If the word *had* is stored right-justified with this instruction:

calc n1 ← "had"

the computer stores it in *n1* as a binary number:



If you request the numerical representation of the word *had* with the -showo- instruction, you will receive its octal equivalent:

00 00 00 00 00 00 00 10 01 04

Thus, the -showo- instruction is useful when you are debugging a program. With practice you will become accustomed to interpreting octal numbers as characters; you can see if the correct characters were stored by requesting the numbers used to store them with the -showo- instruction.

An understanding of this storage system and a knowledge of how much space your characters require are also useful when you are *segmenting* integer variables; integer variables can be divided to hold more than one value. This process is discussed in activity 3-C.

CHECK YOUR UNDERSTANDING

Directions: Answer the following questions by filling in the blanks.

1. How many binary bits does the storage of the character string *Thomas* require?

2. Use the internal codes given in *aids* to determine the octal character or internal codes of the characters in *Thomas*.

T _____ h _____ o _____ m _____ a _____ s _____

3. What is the binary number used to store the character string *Thomas*?

4. What instruction could you use to store the character string *Thomas* on the right side of a variable? _____

On the left side? _____

5. What instruction could you use to display the stored character string *Thomas*?

6. What instruction could you use to display the octal representation of the character string *Thomas*, if it is stored in variable *n1*?

Answers

1. $7 \times 6 = 42$ bits
2. $T = o7024$
 $h = o10$
 $o = o17$
 $m = o15$
 $a = o01$
 $s = o23$
3. $\overbrace{1110000010100}^T \overbrace{001000}^h \overbrace{001111}^o \overbrace{001101}^m \overbrace{000001}^a \overbrace{010011}^s$
4. `calc n1←"Thomas" $$ any integer variable`
`calc n1←'Thomas' $$ any integer variable`
5. `showa n1 $$ integer variable used to store string`
6. `showo n1`

SUMMARY

The two types of variables discussed in this activity are floating point (*v*) variables and integer (*n*) variables. Floating point variables allow for the storage of decimal fractions and very large numbers. Integer variables can be used to store integers and numerical representations of characters; they should not be used to store decimal fractions.

To store a character string, use single quotes to store the characters of the left side of the variable. Use double quotes to store them on the right side of the variable.

Six binary bits are reserved for each lowercase character stored in a variable. Twelve bits are reserved for each uppercase character. Each character of the keyboard has been assigned an internal code. These codes can be seen by requesting the TUTOR feature *internal codes* in the *aids* lesson. When converted to binary numbers, these codes show the binary number the computer uses to store the character string. The `-showo-` instruction will show the octal representation of character strings.



3-C. SEGMENTED VARIABLES

Identify correct ways to segment variables and reference segmented variables.

Consider a case in which you wish to store the test scores of seventy-five students to obtain an average score. You could use seventy-five of your 150 student variables; this method, however, may not leave you enough variables to store other records you wish to keep. Thus, you may want to *segment* your variables to store more than one number in each variable. This activity explains how to segment variables to make better use of variable storage space.

THE SEGMENTATION PRINCIPLE

Activity 1-B explains that each variable can store up to sixty binary digits; when sixty digits are not required, an integer variable can be divided or *segmented*, and different numbers or character strings can be assigned to each segment. This segmentation process greatly increases the number of records that can be stored. Each segment can be thought of as a totally separate variable.

For example, consider the storage of the student test scores. Assume the scores can range from 0 to 100. The decimal number 100 is equal to 144 in octal. 144_8 is equal to 1100100_2 in binary. Thus, the largest score requires only seven bits of storage space; the other fifty-three bits are empty.

The point is, one integer variable has enough space for *eight* scores ($60 \text{ bits} \div 7 \text{ bits per score} = 8 \text{ scores plus a fraction}$). If all student variables required for this record-keeping function are segmented, only ten variables ($75 \text{ scores} \div 8 \text{ scores per variable} = 9 \text{ variables plus a fraction} = 10$) rather than seventy-five are needed.

The general format for segmenting variables is:

```
define segment, SEGMENT NAME=VARIABLE, BITS PER SEGMENT
```

Segment size can only be defined with integers. For example,

```
define segment, score=n1, 7
```

divides variable *n1* and all subsequent variables in the user variables into eight segments of seven bits each, with four bits left over. The name of the segmented variable is *score*.

When you are defining segments, an argument (*s*) may be used to reserve a bit in each segment for the sign of the number. For example,

```
define segment,score=n1,7,s
```

would reserve a bit in each segment of the sign in case negative scores might be stored. Because one bit is reserved for the sign, only six bits are left to represent the number. Therefore, the smallest number that can be stored is -63 and the largest is +63. In order to store scores up to + or -100, you would have to use more variables. If only positive numbers are stored, no sign argument is necessary.

Segmentation by sixes is useful for character manipulation because each character requires six bits of storage space. Ten lowercase characters or five uppercase characters can be stored in a variable. Segmenting that variable allows you to access each character individually.

Segmentation by ones is useful in true/false situations, where 0 and 1 are sufficient to represent the truth value.

REFERENCING SEGMENTS

To use any given segment elsewhere in your program, reference it by name and location number (index). In the score example, the segments would be named as follows:

n1	7 bits	7 bits	7 bits	7 bits	7 bits	7 bits	7 bits	7 bits	4 bits
	score(1)	score(2)	score(3)	score(4)	score(5)	score(6)	score(7)	score(8)	
n2	7 bits	7 bits	7 bits	7 bits	7 bits	7 bits	7 bits	7 bits	4 bits
	score(9)	score(10)	score(11)	score(12)	score(13)	score(14)	score(15)	score(16)	

Thus, the instruction

```
show score(1)
```

would show the leftmost group of seven bits.

Variables may be named and used as full words even though they have been segmented. For example, *n1* in the previous sample could be redefined:

```
define index = n1
```

USING VARIABLES AS THE INDEX

Often the contents of segmented variables are similar in some way. For example, a variable may hold all scores, or all truth values for a series of similar questions. In such situations, you can save time and code by letting one instruction do the work of many; use a variable as the index. Assume, for example, that you wish to track a student's completion status on four parts of a lesson as follows:

- 0 = student has not entered the part
- 1 = student has entered the part, but has not completed it
- 2 = student has completed the part with an unsatisfactory score
- 3 = student has completed the part with a satisfactory score

You might choose to segment variable *status* by twos to hold the values that represent the completion status:

```
define segment,status=n50,2
```

You must then zero the segments of status using one of five methods:

Method 1: Zeroes each segment individually

```
zero    status (1)
zero    status (2)
zero    status (3)
zero    status (4)
```

Method 2: Uses a -calc- to set a value of 0 for each individual segment

```
calc    status (1) ← 0
        status (2) ← 0
        status (3) ← 0
        status (4) ← 0
```

Method 3: Uses a different form of the -calc- instruction to zero each segment

```
calc    status (1) ← status (2) ← status (3) ← status (4) ← 0
```

Method 4: Uses a variable as the index, and assigns a value to this variable in the -doto- instruction

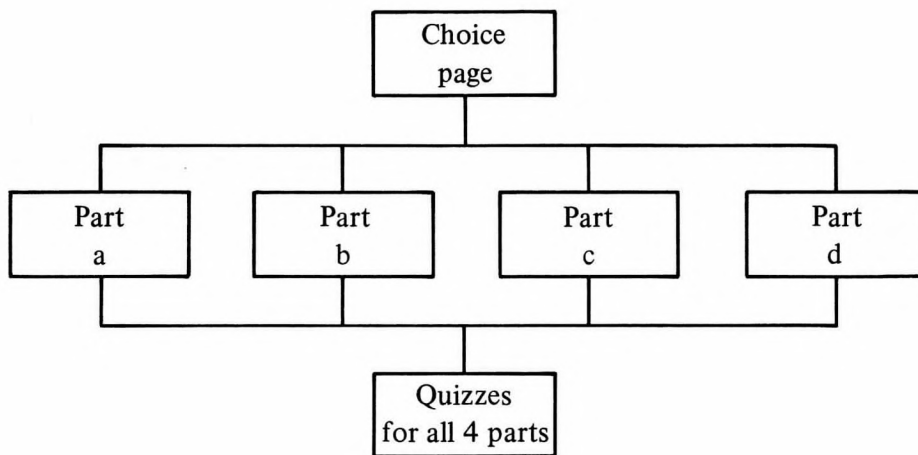
```
doto    1zero,work1 ← 1, 4
calc    status (work1) ← 0
1zero
```

Method 5 (if it is acceptable to zero the entire segmented variable)

```
define segment,status=n50,2
      wstatus = n50
zero   wstatus
```

If you use the fifth method, you are *double defining* variable *n50*. In the first case, it is segmented; in the second case, it is not. Thus, the entire variable can be zeroed (all segments) by zeroing *wstatus*.

The format of this lesson can be diagrammed as follows:



The quizzes for the four parts of the lesson have similar formats—four multiple-choice questions in each quiz. Therefore, you need only one unit for your quizzes; this unit will contain -writec- and -answerc- instructions, conditional on the value of a -match- instruction and the question number. The student must answer three of four questions in each quiz correctly on the first try to receive a satisfactory rating.

Because a -match- instruction will be used to identify the chosen area, this expression is used in the sample code to identify a particular question in a particular area:

(area x 4 + questno)

↑ ↑ ↑

 number of the question in a particular quiz

 number of quizzes

variable in the -match- instruction

The following code contains the essential instructions for recording the students' status:

```

define segment, status=n50, 2
      wstatus = n50
      score    = n51
      questno  = n52
      area     = n53
zero   wstatus
zero   score
***
unit   choice
at     1010
write  Which lesson do you want to take?
      a.  addition
      b.  subtraction
      c.  multiplication
      d.  division
arrow  1620
match  area,a,b,c,d
endarrow
calc   status(area+1)÷1
jump   area-1, aarea, barea, carea, darea
:
unit   aarea
:
unit   barea
:
unit   carea
:
unit   quiz
doto   1ques, questno÷1, 4
at     LOCATION OF YOUR CHOICE
†writec (area×4+questno)÷QUESTION1÷QUESTION2÷...÷QUESTION16
arrow  LOCATION OF YOUR CHOICE
answer (area×4+questno)÷ANSWER1÷ANSWER2÷...÷ANSWER16
endarrow
calcs  ntries=1, score÷score+1,,
1ques
calcs  score>2, status(area+1)÷3, 2
next   choice

```

†If the student were in area *a* (which has a value of 0), and should receive question one, the formula would be evaluated as $(0 \times 4 + 1)$.

Notice, in *unit choice*, that the segment variable *status* uses the value held in the variable *area*. Because *area* has a potential value of zero, +1 is added to its value. If the instruction is

```
calc      status (area)  $\leftarrow$  1
```

and the student presses *a*, the instruction will become:

```
calc      status (0)  $\leftarrow$  1
```

This is illegal. Segment names must have a value of 1 or greater.

As it stands, the instruction

```
calc      status (area+1)  $\leftarrow$  1
```

stores a value of 1, to show that the student has started a particular part but has not completed it.

The student then enters a unit dependent on the part chosen. From this unit, the student is sent to *unit quiz*. Here, four questions are given in a -doto- loop that contains a -calcs- instruction; the -calcs- instruction keeps track of the number of questions the student answered correctly on the first attempt. When a question is answered correctly on the first attempt, the variable *score* is incremented by 1.

The second -calcs- instruction records the student's final completion status after the -doto- loop is completed. If the student's score is greater than 2, the value 3 is stored in the correct segment of variable *status*; this means that the student has completed that particular quiz with satisfactory results. If the student's score is not greater than 2, the value 2 is stored, to show that the quiz was completed with unsatisfactory results.

In summary, the use of the variable *area* as an index for segmented variable *status* makes coding and tracking easier. Remember, however, that the index variable must never hold a value of 0; this is one of the errors most frequently made in using variables as indexes in a segmented variable.

CHECK YOUR UNDERSTANDING

Directions: Write the letter of the correct alternative in each blank.

- _____ 1. Which of the following variables should *not* be segmented?
- n5
 - v6
 - v8
 - n10
- _____ 2. If you want to store scores up to 1023 (this requires ten bits per score), what instruction should you use? (All scores are positive.)
- `define segment, n1, 10, s`
 - `define segment, score=n1, 10`
 - `define segment, score=n1, 10, s`
 - `define segment, score=v1, 10`
- _____ 3. If you want to store both positive and negative numbers between -15 and +15 (15 requires four bits), what instruction should you use?
- `define test=n1, 4, s`
 - `define segment, test=n1, 5`
 - `define segment, test=n1, 5, s`
 - `define segment, n1, 5, s`
- _____ 4. In storing characters, it is often useful to segment by ____.
- tens
 - sevens
 - sixes
 - ones
- _____ 5. In storing true/false values, it is often useful to segment by ____.
- sixes
 - fives
 - twos
 - ones
- _____ 6. If a segmented variable named *value*, segmented by sixes, is used, what reference should be used to retrieve the rightmost segment in the second word (variable)?
- `value(1)`
 - `value(8)`
 - `(1)value`
 - `value(20)`

Answers

1. b,c; floating point (v) variables should not be segmented.
2. b; answer *a* is unsatisfactory because it does not name the variable and because a sign argument is not required. Answer *c* is unsatisfactory because of the sign bit, which takes up a space and leaves only nine bits for the number. Answer *d* is unsatisfactory because of its floating point variable.
3. c; a sign argument is required, and five bits will be necessary—one for the sign and four for the number.
4. c; each character requires space for six bits.
5. d; true/false situations can be represented by 1 or \emptyset . They require only one bit.
6. d; a variable segmented by sixes has ten segments, with the rightmost segment of the first word referenced by the name *value(1 \emptyset)*, and the rightmost segment at the second word referenced by the name *value(2 \emptyset)*.



3-D. STRETCHING YOUR VARIABLES

Identify the effects of the reassignment of values and the use of system-defined variables on storage space.

One of the challenges of programming is the task of making the best use of variable storage space. Activity 3-C explains the segmentation process; this activity explains two more ways to stretch the variable storage space available to you. The first way involves reassigning values to variables when the old values are no longer needed. The second way involves making use of system-defined variables, when appropriate.

REASSIGNING VALUES

Many times you will require a variable to store a value for a short time, after which the value is never used again. For example, once a `-doto-` loop has been executed, you may never again require the values generated by that loop.

In these cases, you can reassign values to your variables. You can simply write over the old values or you can redefine or double-define variables. When a second value is placed in a variable, the first value is overwritten, and the lesson continues, using the new value.

Some programmers, when they know a program will require many variables, allow for reassignment by establishing a set of variables in their IEUs that they assume will be overwritten during the lesson. These variables are often defined as *work* variables; they can be assigned any name, but by using the same name every time, you can set a convention for all your lessons and make reassignment less confusing. The following are ways in which work variables can be assigned:

```
define  work1    = v1
        work2    = v2
        work3    = v3
```

or

```
define  wk1      = v1
        wk2      = v2
        wk3      = v3
```

or

Any defined name you want to use

When you double-define a variable, you assign more than one name to the variable. If you used this instruction,

```
define limit    = v1
      loop      = v1
```

it could be assumed that you assigned a value to variable *v1* using the name *limit* in some part of your program, and that when this value was no longer required, you assigned a new value using the name *loop*. Thus, the names of the variables still give you a clue to their purpose, but you can use each variable more than once.

SYSTEM-DEFINED VARIABLES

PLATO Author Language: Part I explains the concept of system-defined variables; they are set by the system, and cannot be defined by the programmer. You have read about or used several such variables, including *ntries*, *where*, *wherex*, and *wherey*. There are many more. If you use these variables when appropriate, you can save your user variables for other purposes. Because you do not have to define system variables, they also save you time.

This section presents a few more of the many system variables available, including *judged*, *anscnt*, *key*, *clock*, *jcount*, *baseu*, and *mainu*. For other system variables, request the TUTOR feature *system reserved words* in lesson *aids*.

System Variable judged

The system variable *judged* stores -1 after an *ok* judgment (if instructions such as -answer- and -ansv- are matched), 0 after a *wrong* judgment (if -wrong-, -wrongv-, -wrongc-, -ntouchw-, and so forth, are matched) and 1 after a *no* judgment (-no-, -judge- *no*, and so forth). Thus, it is useful in conditional instructions, such as -writec- used after -specs-, to give appropriate feedback depending on the response matched. The combination of -writec- and -specs- is discussed in unit 4.

System Variable ansCnt

The system variable *anscnt* stores the number of answer-judging instructions the system encounters before it finds a match for the student's response. Some of the answer-judging instructions you know at this point are -answer-, -wrong-, -answerC-, -wrongc-, -ok-, -no-, -ansv-, -wrongv-, and -match-. This variable is useful when you want to branch the student on the basis of the judging instruction matched.

System Variable key

The system variable *key* holds the ten-bit keycode of the last key a student presses. To see the keycodes (these are not the same as internal codes) for various keys, request TUTOR features *keynames* and *keycodes* in lesson *aids*. Some keys, such as NEXT and DATA, are stored in *key* so often that they can and should be referenced by name in the instruction. System variable *key* can be used to branch students depending on their key presses.

System Variable clock

System variable *clock* places the current time in seconds in a variable. This is not time as you think of it; it is simply a reference point. Thus, you would never want to use this instruction,

```
show    clock
```

because its value will not have meaning. However, to find out how long a student spent in your activity, you could include this instruction at the beginning of your lesson,

```
define  timein  = v1
        totaltm = v2
calc    timein←clock
```

and this instruction at the end of your code:

```
calc    totaltm←clock-timein
```

If you store *totaltm*, you will have a record of the number of seconds spent in the activity.

This instruction

```
store   totaltm÷60
```

would store the time spent in minutes.

(If you want to show the student the actual time of day established by your PLATO system, use the *-clock-* instruction. See *aids*.)

System Variable jcount

The system variable *jcount* holds the number of internal six-bit character codes of a student response. For example, if the student types *cat*, *jcount* is 3. If the

student types *Cat*, *jcount* is 4. System variable *jcount* is affected by instructions that modify the student response, such as the *-specs-* instruction with a *bumpshift* tag. If a *-specs- bumpshift* were included and the student typed *Cat*, *jcount* would be 3.

If you want to test the results stored in *jcount*, try inserting these instructions in your code:

```
unit      count
arrow     61Ø
ok
endarrow
at        81Ø
show      jcount
next      count
```

Condense your code, and test *jcount* with responses of variable lengths. Insert a *-specs- bumpshift* after the arrow, and test *jcount* with responses containing capital letters. The *jcount* variable can be used to test the length of a student response.

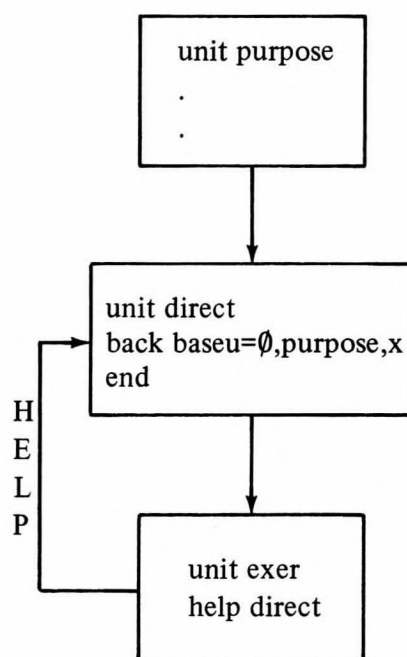
System Variable *baseu*

The system variable *baseu* holds the alphabetic name of the student's current base unit, or a Ø if there is no base unit. This stored information is useful for branching purposes when help sequence units are also used as main units elsewhere in the lesson.

For example, assume that you have included a set of directions to be used in two ways:

- All students will see the directions in the beginning of the lesson.
- Students can press HELP to see the directions *again* while completing an exercise (the directions become a help sequence).

You want to activate the BACK key in the directions unit. If the directions unit is serving as a *main* unit, a BACK key press should take the student back to the purpose of the lesson. If the directions unit is serving as a *help sequence* unit, you want the BACK key to return the student to the exercise. The following diagram shows how the system-defined variable *baseu* can produce the desired results. If *unit direct* is being used as a main unit (if *baseu* = Ø), the BACK key takes the student to *unit purpose*. If *unit direct* is serving as a help sequence unit, (*baseu* ≠ Ø), the BACK key will take the student to *unit exer*.



System Variable mainu

The system variable *mainu* holds the name of the student's current main unit. This is useful for debugging purposes; if you show *mainu* while executing a lesson as a student, you can tell what your main unit is at any time in the lesson. Thus, you will know if you are jumping to the wrong unit. It is also easier to find a unit once you have identified a bug; you can do an *x* search on the unit shown.

To get a better idea of the value of *mainu*, enter this code:

```

unit      main
at        210
write     Is this unit main or auxiliary?
at        510
writet    mainu='main'⚡This is main.⚡This is auxiliary.
***
unit      do
do        main

```

To debug your program using the data in *mainu*, insert this code in the IEU of your program:

```

define    name      = n1 $$ allow two variables
imain     imain
***

```

```

unit      imain
name      name
if        name='YOURS'  $$ put your name here
.         at           101
.         showa        mainu
endif

```

This unit named in the tag of the `-imain-` instruction will be done at the start of every main unit in the lesson. The `-name-` instruction stores the sign-on name of whoever is using the lesson as alphabetic characters in the variables specified in its tag. With the previous code, the main unit will only be shown when you, the author, sign on.

System Variable *ldone*

The system variable *ldone* (the first character is the letter l, not the numeral 1) holds a value of:

- -1 after a `-lesson completed-` instruction has been executed
- 0 after a `-lesson incomplete-` instruction has been executed
- 1 after a `-lesson noend-` instruction has been executed
- 2 if the student has entered but not completed the lesson

Thus, *ldone* can be used to hold a student's completion status. It is mentioned here to make you aware of its existence; with what you know at present, *ldone* is not very useful to you. In the future, however, when you have learned to connect your lessons, you will probably use it often.

The `-lesson-` instructions are used to assign the values. They are routing instructions that are always recognized by the system router (mrouter) but may not be recognized by other routers. For more information on `-lesson-` instructions request TUTOR feature *aids*. Routers are discussed in activity 6-B.

SUMMARY

To reassign values to a variable, you can simply overwrite the old value, or you can double-define the variable and use it in more than one section of your code.

The following table summarizes the information presented on system variables.

<i>Variable</i>	<i>Effect</i>	<i>Possible Users</i>
judged	Stores a: -1 after <i>ok</i> Ø after <i>wrong</i> 1 after <i>no</i>	To give appropriate feedback
anscnt	Stores number of answer-judging instructions encountered before a match is found	Branching based on instruction matched
key	Holds keycode of last key student pressed	Branching based on key press
clock	Places current time in seconds in a variable	Discovering amount of time spent in an activity
jcount	Number of internal 6-bit character codes in student response	Testing length of student response
baseu	Holds alphabetic name of student's current base unit	Branching depending on whether unit is a main unit or help sequence unit
mainu	Holds name of student's current main unit	For debugging
ldone	Holds completion status of lesson if used in conjunction with -lesson- instructions	To store completion records



3-E. AUTHOR-DEFINED ARRAYS

Recognize steps in the processes of defining one-, two-, and three-dimensional arrays.

The word *array*, in programming, represents a method of storing and retrieving information. Usually, information stored in arrays can be thought of as similar pieces of information that require storage in rows, or rows and columns, of variables. The variables or elements of the array can be thought of as compartments. For example, programming the PLATO system to play checkers requires an array of compartments in rows and columns for the checker board; the system must keep track of which compartments of the board hold which player's checkers at any given time.

The activity discusses how to store information in *author-defined* arrays. The programmer defines what will be stored in each compartment or variable. Author-defined arrays can have up to six *dimensions*. One-, two-, and three-dimensional arrays are most often used, and are discussed in this activity.

ONE-DIMENSIONAL ARRAYS

The one-dimensional array is the easiest to use; any piece of information in a one-dimensional array can be called out with only one number because the array consists of only one row or column.

You use a form of a one-dimensional array when you segment integer variables. Any of the segments can be stored or retrieved using one number. For example, the leftmost segment of a segmented variable *score* can be identified as *score(1)*. With author-defined arrays, you define the array in one easy step in your IEU, and use it elsewhere in your code; the important principle to remember is that the pieces of information you store in your array should be used in a similar way, such as in a *-doto-* loop. The following example illustrates the use of a one-dimensional array.

Assume that you are programming a lesson in which:

- Six candidates are interviewed by the student
- Each candidate must be rated on a scale of 1 to 3; decimals such as 2.5 can be used for the ratings

You must store the rating for each candidate and show *all* ratings at the same time later in the lesson.

Because you have six candidates, and because decimals can be used to rate the candidates, six floating point variables must be used to store the ratings. Assume that variables $v10$ through $v15$ will be used. Think of the variables as being stacked in a row:

$v10$
$v11$
$v12$
$v13$
$v14$
$v15$

Previously, you had to define each variable. With an array, you use this format:

```
define VARIABLENAME(X) = V(MATHEMATICAL EXPRESSION)
```

expression defining starting location in terms of the index

any combination of characters will represent the value of the variable

For example, if the value of the first index will be 0, the formula is:

$$\text{variable name (x)} = v(1st + x)$$

If the value of the first index will be 1, either of these two formulas can be used:

$$\text{variable name (x)} = v(1st - 1 + x)$$

$$\text{variable name (x)} = v(1st - 1 + -1)$$

The particular situation determines whether you will set the index at 0 or 1. When using segmented variables, set the index at 1; indexing with 0 results in an execution error because you can't reference $seg(0)$ if seg is a segmented variable. Thus, it is important to analyze your situation carefully before setting your index.

For the interview example, this instruction can be used because the variables in the array are *not* segmented:

```
define canrate(i)=v(10+i)  $$ canrate stands for
*                          "candidate's rating", requires v10-v15
```

The comment is not essential, but it is good coding practice; you can see at a glance which variables are used in your array so you don't use them again. If you want the rating for the first candidate to be stored in $v10$, you can see that the value of this rating must be 0 ($canrate(0)=v(10+0)$). Thus, when coding this lesson, you must ensure that a 0 is used to reference the rating of the first candidate.

The following code contains the essential instructions for coding the interview lesson:

```
define  canrate(i)=v(10+i)  $$ requires v10-v15
        loop      = v16
        cand      = v17
zero    canrate(0),6  $$ initialize to zero because 0
*                                     means candidate has not been
*                                     interviewed
***
unit    call
at      1010
write   Which candidate do you want to rate?
        a.  John
        b.  Sonya
        c.  Ann
        d.  George
        e.  Cynthia
        f.  Harry
match   cand,a,b,c,d,e,f
*   if a is chosen, value in cand is 0; if c is chosen,
*   value in cand is 2; and so forth.
endarrow
jump    cand-1,casea,caseb,casec,cased,casee,dasef
***
unit    rate
next    call
at      2010
write   What rating do you give this candidate?
arrow   2110
:
store   canrate(cand)
:
***
more units of code
***
unit    show
at      1010
```

```

write  Here are the ratings you gave your candidates:
doto   1loop,loop+0,5  $$ six loops for six candidates
*
*                               starting at 0 to ensure that
*                               the first one shown will be v10.
at     1310+loop*100  $$ loop * 100 moves each rating
*                               down one line
writec loop-1+a. John+b. Sonya+c. Ann+...+f. Harry
at     1330+loop*100
show   canrate(loop)  $$ rate for each candidate is shown
***
unit   casea
:
goto   rate
***
unit   caseb
:
goto   rate
***
:

```

TWO-DIMENSIONAL ARRAYS

A two-dimensional array requires two reference numbers for each compartment or variable. It is helpful to think of a two-dimensional array in this way:

	0	1	2
0	0,0	0,1	0,2
1	1,0	1,1	1,2
2	2,0	2,1	2,2
3	3,0	3,1	3,2

Each compartment is referenced by two numbers or indexes—a row number and a column number. The following case study demonstrates the use of a two-dimensional array.

The lesson asks students to evaluate four candidates for credit cards. They are to evaluate the salary, age, and credit rating of each candidate, using a rating scale from 1 to 3. This information can be visually represented as follows, with one variable used to store each rating:

	Salary Rating	Age Rating	Credit Rating
Candidate 1	V10	V11	V12
Candidate 2	V13	V14	V15
Candidate 3	V16	V17	V18
Candidate 4	V19	V20	V21

To write this program, you must first define your array, using these instructions.

```
define slot(i,j)=v(10+3i+j)  $$ requires v10-v21
```

The letter i represents one index on the chart, the candidate being interviewed. The letter j represents the second index on the chart, the area of concern (age, salary, or credit rating). When two-dimensional arrays are being defined, the first index is usually the row, and the second the column.

A particular slot can be filled using the $v(10+3i+j)$ formula in the instruction. The 10 is the first variable in the array. The $3i$ is the number of columns times the number of the candidate. The j is the area number.

For this instruction to work, you must think of your candidates and areas in terms of these values:

```
Candidate 1 = 0
Candidate 2 = 1
Candidate 3 = 2
Candidate 4 = 3

Area Salary = 0
Area Age    = 1
Area Credit = 2
```

This is necessary if the first rating is to be stored in $v10$. The $v10$ variable is reserved for the first candidate's salary area rating. With the values just assigned, the formula would be applied by the system as follows:

$$\text{slot}(i, j) = v(10 + 3 \times 0 + 0)$$

This demonstrates that the value will be stored in variable $v10$ only if the values for i and j are 0.

If you want the values for the rows and columns to start with 1, you must use this formula.

```
define slot(i,j)=v(10+3(i-1)+(j-1))
```

After defining the array, a -zero- instruction is included to zero all variables in the array:

```
zero canrate(0),12
```

Code similar to that used for the one-dimensional array can be used to determine which candidate will be rated:

```
unit    rate
at      1010
write   Which candidate do you wish to rate?
        a.  Jill
        b.  Glen
        c.  Sylvia
        d.  Jeff
match   cand,a,b,c,d
doto    1loop,area←0,2
:
:               QUESTIONS FOR RATINGS IN 3 AREAS
store   slot(cand,area)
:
:
1loop
```

To show the ratings for all candidates in all areas, these instructions can be used:

```
doto    1cand,work1←0,3
at      1310+work1×100
writec  work1-1←Jill←Glen←Sylvia←Jeff
doto    1area,work2←0,2
calcs   work2-1,loc←0,7,14
at      (1325+work1×100)+loc
show    slot(work1×work2)
1area
1cand
```

Notice the use of nested -doto- instruction in this code to enable the candidates and ratings for all three areas to be shown. Notice also the use of the variables *work1* and *work2*. These variables can be replaced by the variables defined as *cand* and *area*; this example is used to illustrate that the indexes can be any variables.

THREE-DIMENSIONAL ARRAYS

Three-dimensional arrays require a third index. The following is an example of a lesson requiring a three-dimensional array.

A company is running a survey on employee satisfaction. The PLATO system will be used to record employee responses to questions. The survey must:

1. Record which of two divisions the employee is in.
2. Indicate the employee's job classification (manager, technical, or clerical).
3. Ask four questions; the same questions are given to each employee, regardless of the job classification. The company wants to see if different job classes respond differently to the same questions.

This information can be diagrammed:

Division 1 (\emptyset)	Management (\emptyset)	Q1 (\emptyset)	v2 \emptyset
		Q2 (1)	v21
		Q3 (2)	v22
		Q4 (3)	v23
	Technical (1)	Q1 (\emptyset)	v24
		Q2 (1)	v25
		Q3 (2)	v26
		Q4 (3)	v27
	Clerical (2)	Q1 (\emptyset)	v28
		Q2 (1)	v29
		Q3 (2)	v3 \emptyset
		Q4 (3)	v31
Division 2 (1)	Management (\emptyset)	Q1 (\emptyset)	v32
		Q2 (1)	v33
		Q3 (2)	v34
		Q4 (3)	v35
	Technical (1)	Q1 (\emptyset)	v36
		Q2 (1)	v37
		Q3 (2)	v38
		Q4 (3)	v39
	Clerical (2)	Q1 (\emptyset)	v4 \emptyset
		Q2 (1)	v41
		Q3 (2)	v42
		Q4 (3)	v43

Notice that the values assigned to each index begin with \emptyset in each case.

The formula for defining a three-dimensional array is similar to that for defining a two-dimensional array, except that another variable or index is added. For this example, the -define- instruction could be:

```
define slot(i,j,k)=v(20+12i+4j+k)
*                               requires v20-v43
```

Because the total number of variables required for each division is 12, the number 12 preceding the index for the division, *i*, will enable records to be stored and retrieved for the correct division. Because each job classification is asked four questions, the number 4 preceding the index for the job classification, *j*, enables records to be stored and retrieved for the correct job classification. The final index, *k*, is reserved for the particular question being asked. Thus, if an employee from division 1, with a clerical classification, were asked the third question in the sequence, the system would apply the define formula as follows:

$$\text{slot}(i, j, k) = v(20 + 12 \times 0 + 4 \times 2 + 2)$$

The employee's answer would be stored in *v30*.

If an employee from division 2, with a management classification, were asked the first question in the sequence, the system would apply the formula in this way:

$$\text{slot}(i, j, k) = v(20 + 12 \times 1 + 4 \times 0 + 0)$$

The employee's answer would be stored in *v32*.

To store records, this instruction could be used in the code:

```
store slot(div,class,quest)
```

To show records, this instruction could be used:

```
show slot(div,class,quest)
```



3-F. SYSTEM-DEFINED AND AUTHOR-DEFINED FUNCTIONS

Identify correct uses of system-defined and author-defined functions.

A function can be thought of as a value dependent on the value of one or more variables. For example, the square root of a number is dependent on the original number; the square root is a *function* of the original number. When yards are converted to feet, the number of feet is a function of the number of yards.

The use of functions can make your programming more efficient. Two types of functions discussed in this activity are *system-defined* functions and *author-defined* functions.

SYSTEM-DEFINED FUNCTIONS

System-defined functions are those used so often by programmers that they have been defined by the system; you can use them without defining them in your code.

Some of the most commonly used functions are included in table 5. Other system-defined functions are listed in *aids*.

TABLE 5
System-Defined Functions

<i>Function</i>	<i>Effect</i>
sqrt(x)	Calculates the square root of a number
round(x)	Rounds numbers. Values $\geq .5$ are rounded up. Values $< .5$ are rounded down.
abs(x)	Returns the absolute value
frac(x)	Returns the value to the right of the decimal point
int(x)	Returns the value (integer) to the left of the decimal point
cos(x)	Returns the cosine of a number
sin(x)	Returns the sine of a number

The number or expression to be evaluated is enclosed in the parenthesis. The function can be part of any calculation or expression.

For example, if you want to store the square root of the number 46529 in the variable *root*, this instruction can be used (*root* has previously been defined):

```
calc    root←sqrt (46529)
```

You may also use a variable in the parenthesis:

```
calc    root←sqrt (choice)
```

As another example, consider a case where you want to determine whether a number is even or odd. You can use the *int* function in an expression:

$$(x \div 2) = \text{int}(x \div 2)$$

If the expression is true, the number *x* is even; if it is false, *x* is odd. This expression can be used any time a conditional test is made.

AUTHOR-DEFINED FUNCTIONS

If you need a function not defined by the system, you can define your own function. Assume that you are writing a program that randomly generates a number of yards and asks students to convert the yards to feet. Yards can be defined in the -define- instruction as a variable:

```
define  yd          = v1
```

The function, *feet*, can then be defined in the -define- instruction as dependent on the number of yards times three:

```
define  yd          = v1
        feet        = yd×3
```

Later in your program, you can use this function as follows:

```
unit    ask
randu   yd,5
at      1010
write   Convert (s,yd) yards to feet.
arrow   1210
specs   noops    $$ forces the student to perform calculation
ansv    feet
endarrow
```

Notice that *feet* is used only as a function; it is not defined as a variable. If it were defined as a variable elsewhere in the code, such as

```
define feet = v2
```

a condense error would occur.

Now consider a function that determines the area of a rectangle, based on the rectangle's base and height. This function can be defined in this way:

```
define base = v1
      height = v2
      area = base*height
```

Up to six arguments can be used in an expression associated with a defined function. The following expression uses two:

```
define height(i,j)=2.54(12*i+j)
```

This defined function can be used to convert feet and inches to centimeters. The *i* and *j* arguments are dummy arguments; their values will be determined later in the program. The *i* argument will hold the number of feet, and the *j* argument will hold the number of inches. When actually used in the lesson, the function can be used as follows. The defined variables *ft* and *in* are representative; any values can be placed in the parentheses:

```
:
:ansv height(ft,in)
:
```

As a final example, the following defined functions take a coarse-grid number and return its *x* and *y* coordinates in fine-grid dots. This is useful when you are writing in coarse-grid, but want to box your writing with fine-grid dots:

```
define finex(i)=800*frac(i÷100)-8
      finey(i)=512-16*int(i÷100)
```

Notice that the system-defined functions *frac* and *int* are used to define this function.

Once you define a function, such as *height*, you may use its label as you would use a variable. Always remember that the function is dependent upon the value of a variable if you used a variable in the definition.

CHECK YOUR UNDERSTANDING**Section 1**

Directions: Write a program that:

1. Randomly generates the base and height of a triangle
2. Asks the student to determine the area of the triangle (area = $(1/2)$ base times height)
3. Evaluates the student response using an author-defined function.

An acceptable solution to this problem is found in the *Answers* section.

Section 2

Directions: Write a program that:

1. Randomly generates decimal numbers between 1.0 and 5.0, with one digit to the left of the decimal point and one digit to the right of the decimal
2. Asks the student to round the number to the nearest whole number
3. Evaluates the student responses using the system-defined function *round* in an -ansv- instruction

This program is more difficult to write than the first. It requires the use of both an author-defined and a system-defined function. These hints will help you write it:

1. This program requires a -randu- instruction with two arguments. The second argument will generate whole numbers. Thus, an author-defined function is necessary to convert the whole numbers to decimal numbers.
2. A new instruction is required to ensure that decimal numbers are always shown to the student. The -showt- instruction serves this purpose. Its format is:

`showt VARIABLE, L, R`

The first argument is the variable or expression to be displayed. The second argument is the number of digits to be shown to the left of the decimal point. The third argument is the number of digits to be shown to the right of the decimal point.

An acceptable solution to this problem is contained in the *Answers* section. When you have coded your solutions to your satisfaction, enter the code in your file space; execute the lesson as a student to determine if it performs as you had anticipated.

Answers

Solution 1:

```
define  base      = v1
        height    = v2
        area      = (1/2) * base * height
***

unit    find
randu   base, 9
randu   height, 9
at      1010
write   Determine the area of a triangle with a base
        of <s,base> and a height of <s,height>.
arrow   1310
specs   noops
ansv    area
endarrow
```

Solution 2:

```
define  random    = v1
        stunum    = (random+9) ÷ 10
unit    round
randu   random, 41
at      810
write   Round the number shown below to the
        nearest whole number.

at      1010
showt   stunum, 1, 1
arrow   1210
ansv    round(stunum)
endarrow
```



3-G. COMMON AND STORAGE VARIABLES

Identify characteristics of common and storage variables.

A house often seems to have enough storage space when you first inspect it, but when you move in, you may find that more closets would be helpful. So it is with variables. You now have student variables and system variables at your disposal; in the future, however, you may find that you need more storage space, or storage space that serves a different purpose.

This activity begins with a discussion of the PLATO storage system and how it accommodates two additional types of storage, *common variables* and *storage variables*. It then presents instructions that can be used to manipulate these variables.

THE PLATO STORAGE SYSTEM

To understand the use of common and storage variables, you should know a few facts about the PLATO storage system. The PLATO system has three main types of memory. *PLATO Author Language: Part I* discusses disk and ECS. Disk is the slowest, but is the most abundant and the least expensive. ECS is faster, but there is less of it, and it is more expensive than disk. Central Memory (CM) storage is the fastest, but there is even less CM available than ECS, and it is the most expensive. It is in CM that the actual processing of a lesson takes place. CM can only process one lesson at a time, but this processing takes place so fast that users are seldom aware that they are “taking turns.”

The common variables discussed in this activity are originally stored on disk. When required in a lesson, a copy of these variables is brought into ECS. Storage variables are locations in ECS; they are not stored on disk.

When a lesson is in use, a copy of the current lesson and a copy of the user variables for a particular user are put in CM. Also in CM is another bank of 1,500 variables. These are called *CM common*. They are referenced in much the same way as user variables, except that the letter *c* is added to the reference. Thus, *nc3* references the third CM common variable as an integer, and *vc 351* references the 351st CM common variable as a floating point number. Because there are 1,500 CM common variables, the numbers in the references must be between 1 and 1,500. CM common variables can have defined names similar to user variables. The following are examples of CM common variables with defined names:

```
define course = nc153
      student = nc285
```

Like floating point and integer variables, storage and ECS common variables occupy the same storage locations when loaded into CM. Thus, no matter how you combine ECS common and storage variables in CM common, you can only have a total of 1,500 in CM at any one time.

Figure 2 shows the relationships, in terms of locations, of the variables used to hold common and storage information.

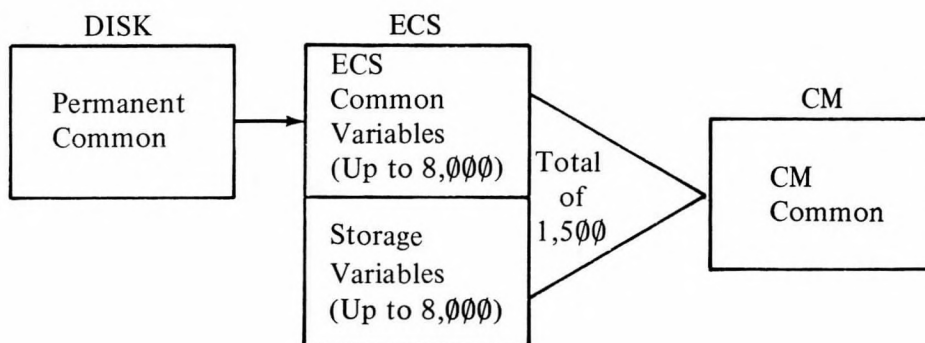


Figure 2. Relationship of variables, showing 16,000 total common and storage variables possible

From this point on, variables used to hold common or storage data will be discussed in terms of their location. For example, an ECS common variable is a location in ECS. A CM common variable is a location in CM. The term *common variable*, without a location qualifier, refers to the common variable concept—a variable that holds cumulative data.

COMMON VARIABLES

Now that you have read about the PLATO storage system, a more detailed discussion of common variables is possible. Common variables are shared by all users. Only one copy resides in ECS, no matter how many people are using the lesson at a given time. Thus, common variables are useful when you must keep cumulative records for all students, such as:

- How many students have completed a lesson
- How many students passed or failed a question on a quiz
- Average student completion time for a lesson

This information can be used to allow students to see how their progress compares to that of other students.

Common variables are also useful for storing information that is common to all students, such as the correct answers for a series of questions. Two types of common variables exist, permanent and temporary. Permanent is perhaps the most useful, and is discussed in this activity. Temporary common variables are not discussed. Permanent common variables reside on disk between sessions. A copy is kept on disk during session, but may not reflect the changes made on the variables. Thus, permanent common keeps a permanent record for all students.

Up to 8,000 words of permanent common can be loaded into ECS. However, only 1,500 can be loaded into CM at any one time. This is the challenge: You must be sure that you load into CM from ECS only those common variables needed at a given time. Instructions are presented later in this activity that enable you to manipulate the transfer of your common variables from ECS to CM and back again.

STORAGE VARIABLES

Storage variables, like student variables, are unique to each user; ECS storage space required for a lesson increases when more people use it.

You can reserve up to 8,000 storage variables. However, storage variables occupy CM common, just as ECS common variables do. When common and storage variables are loaded into CM, common variables are loaded first. If you are loading storage variables into locations already filled with common variables, the common variables will be overwritten; the storage values will replace the common values. The system does not tell you that your locations overlap; you are responsible for ensuring that your common and storage variables are loaded properly.

USING COMMON VARIABLES

Using common variables requires two steps; you must establish a common block, and you must insert a -common- instruction in your code. To create a common block, go to your block display and press the shifted letter of your last block. Choose the option for a common block and name the block. You will then be asked how many words of common you want. If ECS is a prime consideration, you might as well specify 64 because the system "charges" you in multiples of 64 words of ECS.

Later, when you want to see what is stored in your common block, press the letter of your common block, and press NEXT to inspect its contents. To edit your common block, press DATA instead of NEXT. Then choose the *inspect change* option. When you choose this option, you can press HELP for help in

editing. When you are editing in the common block, it is easy to accidentally destroy or change the values in your common variables. Thus, before you edit a common block, unless you are well-versed in common editing, you should create a common block and practice editing. Common editing is an advanced concept not within the scope of this course; it is mentioned here for your information.

You are more likely, at this point, to edit common variables in your lesson. For example, you might want to increment a common variable by 1 every time a student completes your lesson:

```
calc      nc35←nc35+1
          OR
define    people = nc1
          ⋮
calc      people←people+1
```

After you create a common block, you must insert a -common- instruction in your code to access the permanent common. You may insert it anywhere, but the IEU is usually the best place, for editing purposes. You may insert only one -common- instruction per lesson; more than one causes a condense error.

The format for the common instruction is:

```
common LESSON NAME, BLOCK NAME, NUMBER OF WORDS
```

The lesson name may be omitted if the common blocks are in the lesson containing the -common- instruction.

You are now ready to use up to 1,500 common variables anywhere in your code; if your total common length is less than 1,500 variables, they will be loaded into CM automatically. If you have more than 1,500 words of common, or if you have used storage variables and common variables combining for a total of more than 1,500, you must use the -comload- instruction. The -comload- instruction takes a specified string of common variables, and loads it into CM at the time it is needed. The general format for this instruction is:

```
comload CM LOCATION, ECS LOCATION, NUMBER OF WORDS
```

A specific -comload- instruction could be:

```
comload nc100,500,400
```

This loads common variables from ECS into CM. Storage in CM begins with CM location *nc100*. ECS common variable *500* is loaded into *nc100*. ECS common variable *501* is loaded into *nc101*, and so on, until *400* variables have been loaded.

The -comload- instruction should be inserted in your code wherever you want the loading to occur. It remains in effect until another -comload- instruction is executed.

The -comload- instruction can specify up to three strings of variables to be loaded. For example:

```
comload nc1,100,4
        nc5,250,100
        nc105,400,50
```

The first part of this instruction tag loads four common variables starting with ECS variable *100* into CM variables *nc1* through *nc4*. The second part of the tag loads *100* common variables starting with ECS variable *250* into CM variables *nc5* through *nc104*. The third part of the tag loads fifty common variables starting with ECS variable *400* into CM variables *nc105* through *nc154*.

USING STORAGE VARIABLES

Storage variables are created with the -storage- instruction. No storage block is created. The general format of the -storage- instruction is:

```
storage NUMBER OF WORDS
```

The instruction

```
storage 100
```

allows you to use *100* storage variables anywhere in your code.

Only one -storage- instruction is allowed per lesson. The storage variables for each student are set up and zeroed when the student enters the lesson. They are not retained when the student leaves the lesson; they are zeroed again upon reentry.

Usually, when using a -storage- instruction, you must also include a -stoload- instruction. It uses the same general format as the -comload- instruction; it loads the variables into *vc* or *nc* locations in CM.

Be careful that your `-comload-` and `-stoload-` instructions do not overlap. These instructions

```
stoload vc1,200,50
comload nc40,1,1300
```

cause an eleven-variable overlap. The common variables are always loaded first, and are stored in locations beginning with `nc40`. The storage variables are then loaded into `vc1` through `vc50`, and the values in `nc40` through `nc50` are overwritten.

THE `-transfr-` INSTRUCTION

The `-transfr-` instruction (no *e* in `-transfr-`) transfers values from one variable to another. Transfers can be made between any type of variable, including student variables, common variables, storage variables, and CM common variables. The general format for the `-transfr-` instruction is:

```
transfr FROM LOCATION; TO LOCATION; LENGTH
```

Student variables are labeled with the letters *v* and *n*. Common and storage variables are labeled *common*, *ECS location*, or *storage*, *ECS location* when transfers are made between student variables and common or storage variables. For example,

```
transfr v5;common,200;5
```

transfers the values of student variables `v5` through `v9` into ECS common locations `200` through `204`. This instruction

```
transfr n10;storage,50;10
```

transfers the values of student variables `n10` through `n19` into ECS storage locations `50` through `59`.

ECS common and storage labels can be abbreviated with the letter *c* or *s*. These abbreviations can also be used to transfer values between ECS common or storage locations and CM common locations. For example,

```
transfr c,5000;s,100;50
```

transfers the values of ECS common variables 5,000 through 5,049 into ECS storage variables 100 through 149. This instruction

```
transfr c,4000;vc300;25
```

transfers the values of ECS common variables 4,000 through 4,024 into CM locations vc300 through vc324.

The -transfr- instruction works in conjunction with the -comload- and -stoload- instructions, when the values of common and storage variables are being transferred to and from CM common locations. You cannot transfer any values to or from variables in CM locations not loaded with a -comload- or -stoload-. This -transfr- instruction is illegal:

```
comload nc1,100,40
:
:transfr nc20;c,100;40
```

CM common variables nc41 through nc60 have not been loaded. This transfer instruction is legal:

```
comload nc1,100,40
:
:transfr nc20;c,100;20
```

Transfers from student variables to common variables are useful for a number of purposes. Now that you know the -transfr- instruction is available, you should recognize its uses when you encounter them. Here is one way you may not think of: When a programmer executes a lesson as a student in author mode and must shift-STOP off the terminal in the middle of the lesson, the author's student variables are zeroed. Until now, the author would have to begin the lesson over again, which is frustrating during the debugging phase. If the -transfr- instruction is used to transfer the values of the student variables into permanent common variables, the values remain in storage; the author can transfer these back to student variables and continue to execute the lesson upon return to the terminal, retaining values previously set.

CHECK YOUR UNDERSTANDING

Directions: Answer the following questions by filling in the blanks.

1. How many ECS common variables can be stored in ECS?

2. How many storage variables can be stored in ECS?

3. What is the maximum number of common and storage variables that can be loaded into CM?

4. Which type of variable discussed in this activity requires the creation of a special block?

5. Which instruction accesses permanent common?

6. What is its format?

7. What instruction is required if the total common length is greater than 1,500 variables?

8. What instruction creates storage variables?

9. What is the purpose of the -stoload- instruction?

10. What is the purpose of the -transfr- instruction?

Answers

1. 8,000
2. 8,000
3. 1,500
4. Common
5. -common-
6. Common lesson name, block name, number of words
7. -comload-
8. -storage-
9. It loads storage variables into CM common, if more than 1,500 common and storage variables are loaded into ECS.
10. It transfers values from one variable to another.

3-H. DATASETS AND NAMESETS



Identify characteristics of namesets and datasets.

PLATO Author Language: Part I explains that a limited amount of ECS storage space is available for each lesson you program. But what do you do if your lesson requires more than your fair share of ECS? Datasets and namesets are one solution to this problem. They can also be used for a number of other purposes.

This activity explains the general purposes and structure of datasets and namesets. It also presents instructions that can be used to manipulate datasets and namesets. Because datasets and namesets must be set up by an account director, and because your need for them is probably not immediate, this course presents only enough information to make you aware of their existence and general use. If, in the future, you feel that you require a dataset or nameset, you can read further by requesting the TUTOR features *dataset* and *nameset* in lesson *aids*.

THE PURPOSE AND STRUCTURE OF DATASETS

In the past, you've worked with lesson files; when you need a file, you ask your account director for file space for a normal tutor lesson. A dataset is another type of file that is also obtained from the account director. It holds records, but it doesn't calculate or use any PLATO Author Language instructions. You can think of a dataset as a large bank of variables. These variables are stored on disk and do not add to your ECS.

The variables in a dataset are organized into records. These records can be as small as 64 words, or as large as 512 words. The records are stored in parts which, in turn, are divided into blocks, just as with normal file space.

It is easy to determine the number of words available in a one-part dataset if you consider the size of a normal lesson file block. A block can hold up to 320 computer words. Seven blocks make a part, with the first block in the first part containing the directory. Thus, it is easy to determine how many records fit in a part. If the records are 64 words long, the first part of the dataset can hold 30 records. Six blocks \times 320 words per block = 1,920 words in the first part. (Remember that block 0 is reserved for the directory); 1,920 words divided by 64 words per record equals 30 records. If a record is too long for one block, it automatically continues in the next block. You can request datasets of up to 63 parts (140,000 words).

MANIPULATING THE DATASET

The directory in your first dataset part is the same as the directory in a normal file; it is here that the security codes are specified. The read code or the write code in a dataset should match the attach code in your lesson. Both should be set. If you do not establish a code, other people can change your dataset by attaching it and putting in information from their lessons. When you are accessing a dataset in lesson code, the dataset security code must match the normal file attach code; if the codes do not match, the dataset will not be attached.

The `-attach-` instruction is used to attach a dataset to a lesson. The simplest format for the `-attach-` instruction is:

```
attach DATASET FILE NAME
```

You will choose a dataset file name when your account director sets up your dataset file.

The name of your dataset can be contained in an integer variable and used in the `-attach-` instruction; it must be enclosed in parentheses:

```
attach (N VARIABLE)
```

The `-detach-` instruction, which has no tag, detaches the dataset from the lesson. The `-attach-` instruction, however, merely attaches the file. If you want to actually use the file in your lesson, you must bring data from the dataset into ECS with the `-datain-` instruction. Its general format is:

```
datain START RECORD; ECS START LOCATION; # OF RECORDS
```

Records can be brought into ECS common, storage variables, and student variables. They cannot be brought into CM common (`vc1` through `vc15000`) or (`nc`) variables. Thus, the following `-datain-` instructions are legal:

```
datain 14;storage,50;5
datain 1;common,5;20
datain 15;n1;1
datain 2;v5;2
```

The first instruction brings records 14 through 18 into ECS, where they are stored in locations beginning with storage variable 50. The second instruction brings in records 1 through 20 and stores them in locations beginning with common variable 5. The last two instructions store records in integer and floating point user variables.

The number of variables brought into ECS must fit in the ECS variables specified. This instruction

```
datain 2;v5;2
```

is legal as long as the records do not total more than 145 words; more than 145 words cannot be held in the specified floating point variables.

A dataset brought into ECS has one of two statuses. The *read/write* status is set by default; read/write status allows the variables to be changed before they are put back on disk. The *read only* status, which must be specified, allows data to be brought in, but the data cannot be changed.

The -dataout- instruction transfers data from ECS common and storage or student variables to the dataset. Its format is:

```
dataout START RECORD; START LOCATION; # OF RECORDS
```

It takes data from the starting location specified and puts it in records in the dataset, beginning with the record specified.

THE PURPOSE AND STRUCTURE OF NAMESETS

A nameset is a special kind of dataset in which records or sets of records are given names. The names can be up to thirty characters in length, and an additional twenty-four bits are allowed for storing optional variable information. The size of the names and the size of the records are fixed when the nameset is created by your account directory. Like dataset records, nameset records can be from 64 to 512 words long.

To attach a nameset to a file, a form of the -attach- instruction is used:

```
attach NAME OF NAMESET
```

The -datain- and -dataout- instructions are used to transfer the variables.

For more information about naming records in namesets and other actions associated with namesets, see *namesets* in lesson *aids*.



3-I. ARGUMENT PASSING: ANOTHER WAY TO ASSIGN VALUES

Identify correct ways to pass arguments.

In the past, you have used the `-calc-` instruction to assign values to variables. Sometimes, however, an easier method can be applied using a `-do-`, `-join-`, or `-jump-` instruction. This method is called *passing arguments*. Values contained in the `-do-` or `-join-` instructions can be “passed” to the auxiliary unit. Values contained in the `-jump-` instruction are passed to the new main unit. This activity discusses the format for passing arguments and presents a specific application for the process.

THE FORMAT

The general format for passing argument is:

```
do          UNIT NAME (VALUE 1, VALUE 2, ..., VALUE 10)
join        UNIT NAME (VALUE 1, VALUE 2, ..., VALUE 10)
jump        UNIT NAME (VALUE 1, VALUE 2, ..., VALUE 10)
***
unit        UNIT NAME (VARIABLE 1, VARIABLE 2, ..., VARIABLE 10)
```

If you want to assign the values 4, 5, and 6 to variables *n1*, *n2*, and *n3* in auxiliary unit *compute*, these instructions can be used:

```
⋮
do          compute (4, 5, 6)
⋮
unit        compute (n1, n2, n3)
⋮
```

The `-do-` instruction lists the values to be assigned to the variables of the auxiliary unit. The `-unit-` instruction in the auxiliary unit specifies the variables that will receive the values. No `-calc-` instruction is needed.

Any type of expression can be passed. The argumented `-unit-` instruction can specify up to ten variables. The `-do-`, `-join-`, or `-jump-` instruction can specify fewer arguments than the `-unit-` instruction, but cannot specify more. If more arguments are specified, there are no variables assigned to hold the extra argument. If fewer arguments are specified in the `-do-`, `-join-`, or `-jump-` instruction than in the `-unit-`

instruction, variables in the accessed unit that are not assigned values retain any previous values held.

A SPECIFIC APPLICATION

Assume you are coding a lesson that:

- Presents a case study
- Erases the case study
- Presents a problem based on the case study

Ten case studies, of varying lengths, must be presented.

In the past, you had to use ten sets of instructions to perform the erasures. For example:

```

at      1010
erase   35,7
:
at      1210
erase   40,8
:
at      1110
erase   50,10
:

```

This means twenty lines of code in all.

If you pass arguments with a -do- instruction, however, you can make your code more efficient. Use -do- instructions to list the location, width, and number of lines for the erasures:

```

do      erase(1010,35,7)
do      erase(1210,40,8)
do      erase(1110,50,10)

```

Use an auxiliary unit, *erase*, to perform the erasures based on the values assigned to its variables:

```

unit    case1
at      1010
do      case1a
pause   keys=next,term
do      erase(1010,35,7)
do      problem1
:

```

```

***
unit      case2
at        1210
do        case2a
pause     keys=next,term
do        erase(1210,40,8)
do        problem2
:
***
unit      case3
at        1110
do        case3a
pause     keys=next,term
do        erase(1110,50,10)
do        problem3
:
***
unit      erase(wk1,wk2,wk3)
at        wk1
erase     wk2,wk3

```

The variables *wk1*, *wk2*, and *wk3* must be defined in your IEU.

Arguments can also be passed conditionally. For the previous example, you could use a conditional -do- instruction to write the ten case studies, and a conditional -do- instruction for the erasures:

```

define    case      = v1
calc      case←0
***
do        case-1,case1a,case2a,case3a,...,case10a
pause     keys=next,term
do        case-1,erase(1010,35,7),erase(1210,40,8),
           erase(1110,50,10)
do        case-1,problem1,problem2,...,problem10
:

```

4

Response Processing Options

This unit contains a collection of instructions that provide you with more response judging options. It discusses the `-judge-` instruction and several other regular instructions used in arrow processing. It explains additional `-specs-` instruction tags, the use of `-specs-` as a marker, several judging instructions, logical operations, and the `-storen-` instruction.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify the effects of the `-list-`, `-long-`, `-time-`, `-catchup-`, and `-jkey-` instructions
- Identify the effects of the `-judge-` instruction
- Identify the effects of the `-exact-`, `-exactc-`, `-bump-`, `-put-`, and `-putd-` instructions
- Identify the effects of the `-specs-` instruction with *nomark*, *nookno*, *okcap*, and *toler* tags
- Identify ways that the use of `-specs-` as a marker can benefit the programmer
- Identify the effects of the logical operations *\$and\$*, *\$or\$*, and *not(x)*
- Identify the effects of the `-storen-` instruction
- Given programmable-ready material, write a program using instructions presented in this unit

LEARNING ACTIVITIES

- _____ 4-A. More About Arrow Processing: Regular Instructions. Text: This activity discusses the effects of the -list-, -long-, -time-, -catchup-, and -jkey- instructions.
- _____ 4-B. The -judge- Instruction. Text: This activity discusses several forms of the -judge- instruction, which can be used to set judgments and to switch execution states.
- _____ 4-C. More About Arrow Processing: Judging Instructions. Text: The effects of the -exact-, -exactc-, -bump-, -put-, and -putd- instructions are explained.
- _____ 4-D. More About the -specs- Instruction. Text: This activity presents -specs- instruction tags not discussed in *PLATO Author Language: Part I*. It also discusses the use of the -specs- instruction as a marker.
- _____ 4-E. Logical Operations. Text: This activity explains how logical operations can be used in conjunction with logical expressions.
- _____ 4-F. The -storen- Instruction. Text: This activity explains how the -storen- instruction stores student responses and provides practice in its use.
- _____ 4-G. Application Program. Text/Exercise/CAI: This activity provides the opportunity to apply some of the instructions presented in this unit.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of response processing options, you may wish to test out some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

4-A. MORE ABOUT ARROW PROCESSING: REGULAR INSTRUCTIONS



Identify the effects of the -list-, -long-, -time-, -catchup-, and -jkey- instructions.

This activity discusses how arrow processing is affected by five regular instructions: -list-, -long-, -time-, -catchup-, and -jkey-.

THE -list- INSTRUCTION

PLATO Author Language: Part I discusses using lists of synonyms in -answer- and -wrong- instructions. This method is acceptable if the synonyms are used a limited number of times. If, however, you have synonym lists that occur frequently in your lesson, you may want to use the -list- instruction to repeat these synonyms. The format for the -list- instruction is:

```
list      TITLE OF LIST, SYNONYMS
```

For example, you may want to include these synonyms in many instructions of one lesson: *y*, *yes*, *right*, *correct*, *yup*, *yeah*, *correct*, *true*, and *t*. To do so, you can use this instruction:

```
list      plus,y,yes,right,correct,yup,yeah,true,t
```

The command is *list*. The title of the list is *plus*; the title can be no more than seven characters in length, and it cannot be a member of the list. All other words in the list are acceptable synonyms.

The -list- instruction should be placed at the beginning of your lesson in the IEU. You can then include your list of synonyms anywhere in your code using this format (note the double brackets):

```
COMMAND ((TITLE OF LIST))
```

For example, the list of synonyms mentioned previously can be used as follows:

```
list      plus,y,yes,right,correct,yup,yeah,true,t
:
***
```

```

unit    quiz
at      505
write   Was Napoleon I a French emperor?
arrow    805
answer  ((plus))
:
```

The list instruction can also be used to specify optional words. For example:

```

list    option,a,an,the,that,this
***
unit    test
at      507
write   Type the name of a barnyard animal that lays eggs.
arrow    1007
answer  <<option>> (duck,goose,chicken)
```

Note the double brackets.

THE -long- INSTRUCTION

The -arrow- instruction automatically limits the length of a student response to a maximum of 150 characters. The -long- instruction can be used to modify this limit. Its format is:

```
long    NUMBER OF CHARACTERS
```

For example, this instruction

```
long    10
```

will not accept responses of more than ten characters (ten lowercase or five capitals). When the length limit is exceeded, additional characters typed by the student are not shown; the student must either press NEXT to receive a judgment, or erase some characters and enter a response of an acceptable length.

The instruction

```
long    1
```

is an exception to this rule. If this instruction is included, the student will be judged immediately after typing one character; no NEXT key press is required. A -long- instruction with a tag of 1 is sometimes used to select a choice from an index.

The maximum limit for a -long- instruction is 300 characters; after 150 characters are entered, the EDIT key is disabled until instructions are included to prevent this occurrence.

When using the -long- instruction with a tag of 1, you must take care to avoid the frustration caused when students press a letter and immediately press NEXT (the normal method of responding.) The NEXT key press may cause instructions to flash on the screen, because the -long- instruction moves the student automatically into the next unit, or on to regular instructions following a matched instruction; the student's unnecessary NEXT key press moves the student on to still more instructions, or to yet another unit. Also remember that you are overriding normal methods of interaction; for example, the ERASE key cannot be used if the student accidentally strikes the wrong key.

The -long- instruction usually follows an -arrow- instruction and precedes any judging instructions. For example:

```
unit      choice
at        1010
write     Press the letter of the section you want to see:

           a. Causes of the Revolutionary War
           b. Major battles of the Revolutionary War
           c. Famous fighters in the Revolutionary War
arrow     2010
long      1
match     select,a,b,c
endarrow
jump      v1-1,a,b,c
```

THE -time- INSTRUCTION

The -time- instruction can be used if, after a certain length of time, you want to simulate a key press. The PLATO system presses a TIMEUP key, and execution continues.

The general format for the -time- instruction is:

time SECONDS

The following is a sample of coding using the -time- instruction:

```

:
at      1010
write   Here's text.....
time    60
```

```

pause    keys=next,timeup
at       1210
write    Here's more text.....
:

```

In this case, the PLATO system will press the TIMEUP key after sixty seconds, and will break through the pause. The student can also break through the pause with a NEXT key press. If NEXT is pressed, the -time- instruction is no longer in effect; execution continues.

In arrow processing, the -time- instruction can be used to initiate judging after a specified number of seconds. Suppose, for example, that you want to program a typing drill that allows a limited number of seconds for a response. These instructions produce the desired results:

```

unit     black
at       1010
write    blackboard
time     5
arrow    1310
answer   blackboard
ok
***

```

If the student types *blackboard* correctly and presses NEXT within five seconds, the -answer- tag is matched and judging is terminated with an *ok*. There are no regular instructions to be executed following the -answer- instruction. However, if the student types *blackboard* incorrectly, or if the TIMEUP key is pressed before the response is complete, the response will not match the -answer- tag. The -ok- instruction terminates judging with an *ok* judgment to satisfy the arrow so the student can move on to the next word. If you program several units using similar instructions, your timing may be off; the -catchup- instruction, discussed in the next section of this activity, alleviates this problem.

The -time- instruction is only in effect for one judgment, whether the judgment be *ok* or *no*. If the -time- instruction is included before the arrow, subsequent responses after the first incorrect response will not be timed.

A variable can be used in the tag of the -time- instruction. The following code produces a typing drill that allows a limited number of seconds for a response and then goes on to a new word with a new response. In addition, it calculates the number of unsuccessfully typed words:

```

define   wrong      = n1
         seconds    = n2
         loop       = n3
***

```

```

unit      time
calc      wrong←0
doto      1type,loop←1,5
at        1010
erase     15
at        1310
writec    loop-2*blackboard*rats*stuff*things*more things
calcs     loop-2,seconds←5,3,4,4,7
catchup   $$explained in next section
time      seconds
arrow     1310
answerc   loop-2*blackboard*rats*stuff*things*more things
ok
calc      wrong←wrong+1
endarrow
1type

```

THE -catchup- INSTRUCTION

With the -long- instruction with a tag of 1, or the -time- instruction, the -catchup- instruction is often needed. It allows the terminal to "catch up" with what should be displayed before additional instructions are executed. The -catchup- instruction is necessary for the typing drill program so the student is allowed the full number of seconds to respond *after* the word has been displayed.

If this code using the -long- instruction is used

```

:
arrow     1010
long      1
answer    a
wrong     b
write     Sorry, you selected the wrong answer.
:

```

and the student responds by pressing *b* and automatically pressing NEXT, the feedback following the -wrong- instruction will be displayed and immediately erased; the PLATO system assumes that the NEXT key press is that which should erase the feedback, and will return to await a new response at the arrow. A -catchup- instruction following the -write- instruction causes the system to ignore the NEXT key press until all -write- instruction text has been displayed, thus decreasing the probability of student frustration.

THE -jkey- INSTRUCTION

With the instructions you have used thus far, with the exception of the -long- instruction with a tag of 1, the student must press the NEXT key upon entry of a response to enable the PLATO system to judge the response. The -jkey- instruction specifies keys, in addition to NEXT, that will initiate judging. Its general format is:

j key KEY

Usually only function keys can be used in -jkey- instructions. Function keys are those with internal codes (listed in *aids*) greater than 0200.

The -jkey- instruction must follow the -arrow- instruction and precede the first judging instruction. In the following code, the -jkey- instruction is used to take students to different parts of a lesson, depending on their key press. (The -jkey- instruction is usually used in conjunction with conditional instructions.)

```
define person=n1
unit    intview
at      1010
write   David, Joan, and Steve are waiting in the lobby.
        Type in the name of the person you wish to
        interview. Then press NEXT to ask the person to
        come in, or DATA to ask for the files.

arrow   1510
specs   bumpshift
jkey    data
match   person,david,joan,steve
endarrow

jump    key=data,getfile,x
at      1810
write   O.K. Here comes
wrotec  person-1* David.* Joan.* Steve.
next    person-1,david,joan,steve
***

unit    getfile
next    person-1,david,joan,steve
at      1010
wrotec  person-1*DAVID'S*JOAN'S*STEVE'S
at      where+1
write   FILE:
***

unit    david
:
```

```

unit    joan
:
unit    steve
:

```

Note the use of the system variable *key* in the -jump- instruction of this code. This variable records the key most recently pressed by the student. The value recorded is a number representing the key. In this case, the number representing the DATA key is stored if the student presses DATA, causing the student to jump to *unit getfile*. If the NEXT key is pressed, the student “falls through” to the instructions following the -jump- instruction.

SUMMARY

The following chart summarizes the information presented on regular instructions used in arrow processing.

<i>Command</i>	<i>Tag(s)</i>	<i>Effect</i>
list	title, synonyms in code: ((title)) <<title>>	Repeats of list of synonyms
long	number of characters	Sets new limit for the number of characters in a student response
time	seconds variable	PLATO system presses the TIMEUP key after a specified number of seconds, after which execution continues
catchup	blank	Allows the terminal to “catch up” on displays before the -time- and -long- instructions are executed
jkey	key	Specifies keys that will initiate judging in addition to NEXT



4-B. THE -judge- INSTRUCTION

Identify the effects of the -judge- instruction.

When the PLATO system encounters a judging instruction after an -arrow- instruction, it changes to judging state. It continues in judging state until it encounters a judging instruction that switches it to regular state. It then executes those regular instructions following the matched judging instruction, and preceding other judging instructions in the arrow processing.

Once the switch has been made to regular state, the -judge- instruction can be used to “reopen” judging. It can also set the judgment, which is particularly useful when you want to reverse a judgment made by a judging instruction. The effect of the -judge- instruction depends on the tag used. This activity discusses two categories that comprise most -judge- instruction tags:

- Those that throw the system back into judging state (reopen judging)
- Those that set judgments

The one exception to this categorization is the -judge- instruction with a *quit* tag; this tag is not discussed in this activity.

Many programmers, when they first encounter the -judge- instruction, think of it as a judging instruction. It is *not*. It is a regular instruction and is only executed in regular state. If the system is in judging state, the -judge- instruction is ignored. The -judge- instruction is only executed after an ok or no judgment has been set.

REOPENING JUDGING

Four tags that can be used with a -judge- instruction to reopen judging are discussed in this activity. These tags are *ignore*, *rejudge*, *continue*, and *exit*.

The ignore Tag

The -judge- instruction with an *ignore* tag causes the PLATO system to erase (ignore) the student response and return to the arrow in judging state to wait for a new response; it reopens judging. For example, consider a multiple-choice question with three acceptable choices, *a*, *b*, and *c*. There is no need to provide feedback for any other key presses. Unacceptable key presses should be ignored, and the student should be required to press an appropriate key.

These instructions do *not* achieve the desired results:

```
match    correct,a,b,c
endarrow
```

If the student should press the letter *d*, or any unacceptable letter, a *no* judgment is given; the student must press the NEXT, ERASE, or EDIT key to clear the response and try again.

If, however, a -judge- instruction is used conditionally with an *ignore* tag, the desired results are obtained:

```
match    correct,a,b,c
judge    correct,ignore,x
endarrow
```

The -match- instruction sets *correct* to -1 for any key press other than *a*, *b*, or *c*, and switches execution to regular state. The regular instruction -judge- is then executed. Used conditionally, it causes a fall through if the variable is greater than -1. If the variable is -1, the response is ignored; it is erased immediately upon entry with a NEXT key press, and a new response is required.

The rejudge Tag

Unlike the *ignore* tag, the *rejudge* tag does not require a new response, and there is no return to the arrow. It switches the system to judging state and subsequent judging instructions are executed; these instructions evaluate the *original* response. This tag can be used if, for some reason, you want to make a judgment on a response, and then reopen judging and *rejudge* the original response. For example, assume that a student enters the response *Dalmatian*. If you include a -specs-bumpshift, the modified response *dalmatian* will be compared to the subsequent answer-judging instructions. If you want to *rejudge* the original response (*Dalmatian*) with another set of answer-judging instructions, a -judge- instruction with a *rejudge* tag will reopen judging and allow rejudgment of the original response.

The continue Tag

The *continue* tag is similar to the *rejudge* tag in that it also reopens judging; unlike the *ignore* tag, it does *not* cause the system to return to the arrow. However, unlike the *rejudge* tag, the *continue* tag uses the response as *modified* by the -specs- instruction or other response modification instructions. For example, the modified response *dalmatian* could be rejudged by a second set of answer-judging instructions.

The exit Tag

The `-judge-` instruction with an *exit* tag rescinds judgment and waits for further keys of the response. In the following example, students are taking a typing lesson. If they make a mistake when typing, they must retype the word from the beginning; they cannot erase the mistake and continue typing. If a student does try to correct the response, the `-jkey-` instruction causes the ERASE, shift-ERASE, EDIT, and shift-EDIT keys to enter the student response instead of serving their normal purposes. The response is then judged *no* because it does not match the `-answer-` instruction tag; however, the `-judge-` instruction with an *exit* tag rescinds the judgment immediately; the student never sees the *no*. This is appropriate, as the student obviously knows that a mistake has been made; the *exit* tag is executed only if the student presses EDIT or ERASE. The *exit* tag forces the student to continue typing (finish the word, with error present) or request judgment with a NEXT key press. The system treats the continuation as characters added to the old response. When the word has been typed correctly, the `-judge-` instruction is never executed because the `-no-` precedes it. (Only regular instructions following the matched judging instruction and preceding the next judging instruction are executed.) When the word has been typed (uncorrected) and entered with a NEXT key press, the `-judge-` instruction allows the student to fall through to the `-endarrow-` instruction, and a normal *no* judgment is given:

```
unit      type
at        1010
write     philanthropist
arrow     2010
jkey      erase,erase1,edit,edit1
answer    philanthropist
no
judge     key=next,x,exit
endarrow
```

SETTING JUDGMENTS

When used for setting judgments, the `-judge-` instruction is often used conditionally. Four tags are used to set judgments: *no*, *ok*, *okquit*, and *noquit*.

The *ok* and *no* tags simply set the judgment; they do not switch the system's execution state. They cause the system to set the judgment and to continue in regular state; it executes subsequent regular instructions in the arrow. If the system encounters a regular instruction such as `-judge-` with a *continue* tag, the state can be switched at this point, and judging can be reopened.

The following is a sample of code that uses one `-judge-` instruction to switch execution states and another to set a judgment. In this example, the student is to

select cases from an index. The cases must be chosen by selecting the number 1, 2, or 3. Any other key presses are ignored, and the student must reenter with an appropriate key press.

Once an appropriate key press has been made, the second -judge- instruction can be executed. The variable *casedn* contains the status of the case. If the status is \emptyset , the case has not been completed. The system falls through with the *ok* judgment set by the -match-. If the status is other than \emptyset , meaning that the case has already been completed, the judgment is changed to *no* and the student is asked to choose a case that has not been completed.

```
define case = v1
      casedn(i)=v(1+i) $$requires v2,v3,v4
zero   casedn(1),3
:
arrow  2010
match  case,1,2,3
judge  case,ignore,x
judge  casedn(case+1)≠0,no,x
writec casedn(case+1)≠0*Please choose one that you have
      not yet done.*
endarrow
```

The *okquit* and *noquit* tags set judgment to *ok* or *no*, stop processing regular instructions following the -judge- instruction, and return the system to the -specs- instruction (if any). Regular instructions following the -specs- instruction and preceding the first judging instruction are then executed. Processing regular instructions following a -specs- instruction is not unique to the *okquit* and *noquit* tags of the -judge- instruction. This concept is explained more fully in activity 4-D. If no -specs- is included and the tag is *noquit*, the system returns to the first judging instruction; the student must clear the response and enter a new response. If no -specs- is included and the tag is *okquit*, arrow processing is complete. The system proceeds to the next -endarrow-, -arrow-, and -unit- instruction.

SUMMARY

The following chart summarizes the information presented on the -judge- instructions.

<i>Tag</i>	<i>Category</i>	<i>Effect</i>
ignore	Reopens judging	Causes the system to ignore the student response and return to the arrow to await a new response
rejudge	Reopens judging	Switches system to judging state; subsequent judging instructions evaluate the <i>original</i> response
continue	Reopens judging	Switches system to judging state; subsequent judging instructions evaluate the <i>modified</i> response
exit	Reopens judging	Rescinds judgment and waits for further keys of the response
no	Sets judgment	Causes system to set a <i>no</i> judgment and continue in regular state
ok	Sets judgment	Causes system to set an <i>ok</i> judgment and continue in regular state
noquit	Sets judgment	Sets judgment to <i>no</i> , stops processing regular instructions following the -judge- instruction, and returns the system to the -specs- instruction, if any (If no -specs- is included, the system returns to the first judging instruction and waits for the response to be cleared)
okquit	Sets judgment	Sets <i>ok</i> judgment, stops processing regular instructions following the -judge-, and returns the system to the -specs- instruction, if any (If no -specs- is included, arrow processing is complete)



4-C. MORE ABOUT ARROW PROCESSING: JUDGING INSTRUCTIONS

Identify the effects of the `-exact-`, `-exactc-`, `-bump-`, `-put-`, and `-putd-` instructions.

Thus far, you have used the `-specs-` instruction to modify a student response, and various instructions to judge the response. This activity explains five additional instructions that allow you to modify and judge responses: `-exact-`, `-exactc-`, `-bump-`, `-put-`, and `-putd-`.

THE `-exact-` INSTRUCTION

Assume you are programming a lesson such as "Letter Writing," in which the punctuation and spaces in a student response are very important. With the judging instructions you know, punctuation and spaces are ignored in the student response and are used as delimiters in the code.

For example, if you are programming on letter writing, you might ask a student to type the greeting of a letter in the way most commonly used. You want the student to type *Dear John*, and you want the PLATO system to judge the capitalization, the space, and the comma. An `-answer-` instruction will not judge the comma; it will be ignored in the student response. In situations like this, when the student must give a response exactly as you have specified, use the `-exact-` instruction. This instruction

```
exact    Dear John,
```

requires the student to enter the response exactly as you have coded it. No automatic markups are provided for incorrect student responses when the `-exact-` instruction is used.

The `-exact-` instruction can also be used with a blank tag. If used in conjunction with `-judge-` instruction with an *ignore* tag, the `-exact-` instruction will cause a NEXT key press to be ignored. The `-exact-` and `-judge-` instructions are combined as follows:

```
exact
judge    ignore
```

THE -exactc- INSTRUCTION

The -exactc- instruction is the conditional form of the -exact- instruction. Semi-colons can be used as delimiters when the student response must contain a comma. For example:

```
exactc v1;Dear John,;Dear Mary,;Dear Fred,
```

Commas can also be used as delimiters:

```
at      1010
write   Change this fraction to a percent. Be sure to
        include the percent sign.
writec  v1+2/10+3/5+6/10+4/5
arrow   1308
exactc  v1,20%,60%,60%,80%
```

No automatic markups are provided for incorrect responses when the -exactc- instruction is used.

THE -bump- INSTRUCTION

The -bump- instruction deletes specified characters from the judging copy of the student response. The modified copy is then compared to the tags of subsequent judging instructions. For example, if you want the student to convert cents to dollars and cents, you might use these instructions:

```
⋮
arrow   1010
bump    $
exactc  v1,1.00,1.75,2.30
⋮
```

In this example, the student can enter the correct amount with or without the dollar sign.

If the -bump- instruction is used, the characters removed from the judging copy must not be in the tags of any of the following judging instructions. If you "bump" one shifted letter, for example, all shifted letters in the student response will be bumped. If shifted letters are included in judging instruction tags, the PLATO system will not find a match. If a specific letter is "bumped," all occurrences of that letter will be bumped, and it must not appear in the judging instruction tag. For example, a match will never be found if these instructions are used:

```

bump      M
answer    alayalam

```

Even though the initial capital letter of *Malayalam* has been removed, the second occurrence of the letter *m* will prevent a match; both the shift *and* all occurrences of the letter are bumped from the student response.

The effect of the -bump- instruction is cleared when a new -arrow- instruction is encountered. Within an arrow, the effect is cleared when a -judge- instruction with a *rejudge* tag is executed. A maximum of eight characters can be "bumped"; for more characters, use consecutive -bump- instructions.

THE -put- INSTRUCTION

The -put- instruction replaces characters in a student response so they can be judged using standard instructions. The format is:

```

put      CHARACTERS TO BE REPLACED=REPLACEMENT CHARACTERS

```

For example, if the students must enter a mixed fraction, such as $1 \frac{1}{2}$, they will probably place a space between the whole number and the fractional portion of the number. If an -ansv- instruction with a tag of 1.5 is used to judge the response, the student will be judged incorrect; the student's answer will be interpreted as two different values. If the space is replaced with +, however, the answer is acceptable. These instructions resolve the problem:

```

unit      test
arrow     2010
put       =+
*         ↑ press the space bar for this hidden space
ansv      1.5
endarrow

```

If this code is used, all of the following answers are acceptable:

```

1 1/2
1.5
1+1/2
1.50
1 2/4
3/2

```

The -put- instruction should be used with caution, however. For example, if the student types in the correct answer, and accidentally hits the space bar, the -put- instruction in the previous code will replace the space with a +. The student's response becomes $1+1/2+$ and is judged incorrect; the student may not even realize what the problem is because the offending space is hidden.

THE -putd- INSTRUCTION

The -putd- instruction has an effect similar to that of the -put- instruction. It uses a different format, however. Three delimiters must be used. Any character may be used as a delimiter, but it must always be the first character in the tag. For example, if you want to replace the = signs in a student response with a space, you can use the instruction:

```
putd      ::= :
```

The -putd- instruction is also useful when hidden spaces are involved. For the previous example, this instruction

```
putd      ; ;+;
```

makes it easier to see that the space in the student's response has been replaced.

SUMMARY

The following chart summarizes the information presented on judging instructions used in arrow processing.

<i>Command</i>	<i>Tag(s)</i>	<i>Effect</i>
exact	answer to be entered	Requires students to enter a response exactly as you have coded it; no markup provided
exactc	variable;answer1; answer2; variable,answer1, answer2	Requires students to enter responses exactly as you have coded them, with the answer required dependent on the value of a variable
bump	characters to be bumped	Deletes specified characters from the judging copy of the student response
put	characters replaced= replacement characters	Replaces specified characters in a student response with other specified characters
putd	:character replaced: replacement character: (any character can serve as the delimiter)	Replaces specified characters in a student response with other specified characters

4-D. MORE ABOUT THE -specs- INSTRUCTION



Identify the effects of the -specs- instruction with *nomark*, *nookno*, *okcap*, and *toler* tags.

Identify ways that the use of -specs- as a marker can benefit the programmer.

You have used the -specs- instruction with *bumpshift*, *okspell*, *noorder* and *noops* tags to modify the judging of a student response. This activity presents a list of additional -specs- instruction tags. It also discusses the use of the -specs- instruction as a marker.

OTHER -specs- INSTRUCTION TAGS

This section of this activity lists a number of tags that can be used in a -specs- instruction. These tags are used in the same way as the tags you know.

<i>Tag</i>	<i>Effect</i>
1. <i>nomark</i>	The student response, if incorrect, is <i>not</i> marked up with xs, = signs, and so forth.
2. <i>nookno</i>	The <i>ok</i> and <i>no</i> judgments are not displayed to the right of the student response after a judgment.
3. <i>okcap</i>	Capital letters in a student response are ignored if and only if the letters are not capitalized in the judging instruction. Capitals included in the judging instructions must still be included in the response.
4. <i>toler</i>	Numerical responses within 1 percent of the value specified in the tag of the judging instruction are accepted and judged <i>ok</i> .

Still more -specs- instruction tags are listed in *aids*. You are not required to learn additional tags in *aids* at this time; however, you may wish to look up these tags to see if they are of value to you.

THE -specs- INSTRUCTION AS MARKER

When you used the -specs- instruction in previous activities, it performed a function of which you were not aware. The -specs- instruction *sets a marker*; the

PLATO system "remembers" the line in which the -specs- instruction occurs. When the judging instructions and all regular instructions after the judging instructions have been executed, the PLATO system checks back to see if a -specs-marker has been set. If it has, regular instructions following the -specs- instruction and preceding the first judging instruction are executed.

The following is an example of the benefits of using the -specs- instruction as a marker:

```
unit      whale
help      oil
at        151Ø
write     What does the whale produce that was used for
          lamps in the past?
arrow     181Ø
specs     okspell
writec    judged*That's a whale of an answer. This
          useful product is now used as a machine
          lubricant and as an ingredient in soap.*
          HELP is available.
answer    sperm oil
endarrow
```

In this example, if the -answer- instruction matches the student response, the first alternative in the -writec- instruction is shown. Otherwise, the second alternative is shown. The -specs- instruction used as a marker has eliminated two instructions that would have been otherwise to provide feedback for unanticipated responses:

```
no
write     HELP is available.
```

When several regular instructions that should be used after every matched judging instruction are required, using -specs- as a marker saves even more code.

When the -specs- instruction is needed as a marker but a tag is undesirable, the -specs- instruction with a blank tag can be used.

Occasionally there is a need for two -specs- instructions in an arrow. For example, it is unacceptable to use a -specs- instruction with an *okspell* tag on words such as *slate* and *shale* because the system would interpret them as the same word. Other names of rocks, however, may require an *okspell* tag in -answer- or -wrong- instructions. In such cases, it is acceptable to use two -specs- instructions. When executed, the second -specs- instruction overrides the first; the second instruction then becomes the marker. The following is an example of code that includes two -specs- instructions in an arrow:

```

:
arrow      1510
specs      bumpshift
wrong      shale
wrong      slate
specs      bumpshift,okspell
wrong      (mica,feldspar,conglomerate)
answer     granite
:

```

SUMMARY

This activity presented four additional -specs- instruction tags:

- *nomark* prevents the markup of a student response.
- *nookno* prevents the display of *ok* and *no* judgments.
- *okcap* causes capital letters in a student response to be ignored if the letters are not capitalized in the judging instructions.
- *toler* accepts and judges *ok* those numerical responses within 1 percent of the value specified in the tag of the judging instruction.

The -specs- instruction sets a marker; the PLATO system returns to the -specs- instruction when all judging instructions and regular instructions following the judging instructions are executed. Regular instructions following the -specs- are then executed.

More than one -specs- instruction can be included in an arrow; the last -specs- instruction becomes the marker.



4-E. LOGICAL OPERATIONS

Identify the effects of the logical operations *\$and\$*, *\$or\$*, and *not(x)*.

Consider two situations where you have three-choice indexes. In the first situation, the student must complete the first two parts of the lesson before taking the third part. In the second situation, the student can complete either the first part or the second part or *both* before taking the third part. In these situations, *logical operations* combined with logical expressions can be used to obtain the desired results easily. This activity discusses three logical operations: *\$and\$*, *\$or\$*, and *not(x)*.

THE LOGICAL OPERATION *\$and\$*

Logical operations are used to combine logical expressions (expressions that are either true or false). A statement containing two logical expressions separated by the logical operation *\$and\$* is true only if *both* logical expressions are true. Thus, *\$and\$* can be used in the situation in which students must complete the first *and* second part of a lesson before beginning the third. The following instructions produce the desired result:

```
define choice = n1
      adone = n2
      bdone = n3
zero adone, 2
***
unit select
at 1010
write Which part of the lesson do you wish to take?
      a. The habitat of the Eskimo
      b. The culture of the Eskimo
      c. Quiz .
match choice, a, b, c
judge choice, ignore, x $$ takes care of key presses other
*                          than a, b, or c.
```

```

jump      choice-1,aunit,bunit,x
judge     adone=1$and$bdone=1,x,continue
jump      cunit
no
write     You must finish sections a and b first.
endarrow
***
unit      aunit
:
calc      adone←1
next      select
***
unit      bunit
:
calc      bdone←1
next      select
***
unit      cunit
:

```

With this code, the student's key press causes a value of 0, 1, or 2 to be stored in variable *choice*. The -jump- instruction uses this value. If *a* is selected, the student jumps to *unit aunit*; if *b* is selected, the student jumps to *unit bunit*; if *c* is selected, the student falls through so the system can see if the *\$and\$* requirement in the -judge- instruction has been satisfied. If it has (if the value of the statement is true), the student jumps to *unit cunit*. If it has not (if the value of the statement is false), the student is told to complete sections *a* and *b*.

THE LOGICAL OPERATION *\$or\$*

Two logical expressions separated by the logical operation *\$or\$* are judged true if *one* or *both* of the expressions are true. The *\$or\$* can be used in the situation in which the student can complete the first part, the second part, or both the first and second parts of the lesson before taking the third.

To alter the sample code in the previous section to reflect this change, merely change the *\$and\$* to *\$or\$* and change the -write- instruction to say:

```

write     You must complete section a or b first.
           You may complete both if you wish.

```

THE LOGICAL OPERATION *not(x)*

When the PLATO system evaluates logical expressions, each expression is given one of two values: -1 if the expression is true, and 0 if the expression is false. These values are called the *truth values* of the logical expressions.

Logical expressions can use the logical operation *not(x)* to reverse the true value. If the expression *count=6* is true, then *not(count=6)* is false.

In most cases, the use of *not* in a logical expression can be avoided by rewriting the expression. For example, *not(count=6)* is equivalent to *(count≠6)*. Therefore, you may never have to use *not* in a logical expression. However, if you should need it, it is mentioned here so you know it exists.

SUMMARY

The following chart summarizes the effects of the logical operations *and* and *or*.

<i>Logical Operation</i>	<i>If 1st Logical Expression Is:</i>	<i>If 2nd Logical Expression Is:</i>	<i>Stored Result Is:</i>
\$ and \$	T	T	T
	T	F	F
	F	T	F
	F	F	F
\$ or \$	T	T	T
	T	F	T
	F	T	T
	F	F	F

The logical operation *not(x)* reverses the truth value of an expression.



4-F. THE -storen- INSTRUCTION

Identify the effects of the -storen- instruction.

Thus far, two instructions have been used to store student responses; the -store- instruction stores numerical responses and the -storea- instruction stores character strings in a form that allows them to be shown in their original form. This activity discusses a third instruction, the -storen- instruction, that is also used to store student responses. It explains:

- What the -storen- instruction stores
- How the -storen- instruction modifies the judging copy

THE EFFECT OF THE -storen- INSTRUCTION

Consider a case in which you want the student to convert thirty-six inches to feet. Assume that the student enters the response *3 ft*. The -store- instruction yields a *no* judgment on this response; it does not recognize the characters *ft*. The -storea- instruction stores the entire response as a character string; the numerical portion cannot be used for future calculations. The -storen- instruction searches for only the numeric part of a response, even if the response contains characters. Thus, if the student enters *3 ft*, only the *3* is evaluated and stored. The -storen- instruction removes the number from the judging copy. If a response requires more than one number, use consecutive -storen- instructions—one for each numeric part of the response.

For example, if the student is expected to convert eighteen inches to feet *and* inches, this code could be used:

```
define  ft      = v1
        in      = v2
unit    convert
at      710
write   Express 18 inches as feet and inches.
arrow   1010
storen  ft
write   Use digits for the numbers.
storen  in
write   Enter two numbers.
```

```

ok
judge   ft=1$and$in=6,x,no
writec ft=1$and$in=6*Good for you.*Try again.
endarrow

```

Notice the -ok- instruction in the code. As with the -store- instruction, the -storen- instruction terminates judging with a *no* judgment only if no numeric value can be found. The first -storen- instruction will terminate judging state with a *no* judgment only if no arithmetic expressions are found; if no arithmetic expressions are found, the student will receive the feedback, *Use digits for the numbers*, and will have to enter another response. If the response is *1 ft*, the first -storen- instruction will recognize *1* as a valid number, store it in variable *ft*, remove it from the judging copy, and continue in judging state. The second -storen- instruction will fail, however, because no second number will be found. Judging will terminate with a *no* judgment and the -write- instruction *Enter two numbers*. will be executed. Thus, if the -judge- instruction is to be executed, an -ok- instruction is required to terminate judging state, should the student respond with the required format.

HOW THE -storen- INSTRUCTION MODIFIES THE JUDGING COPY

The -storen- instruction removes the numeric part of the response from the judging copy and replaces it with blanks. Figure 3 illustrates this process for a situation in which two -storen- instructions have been included:

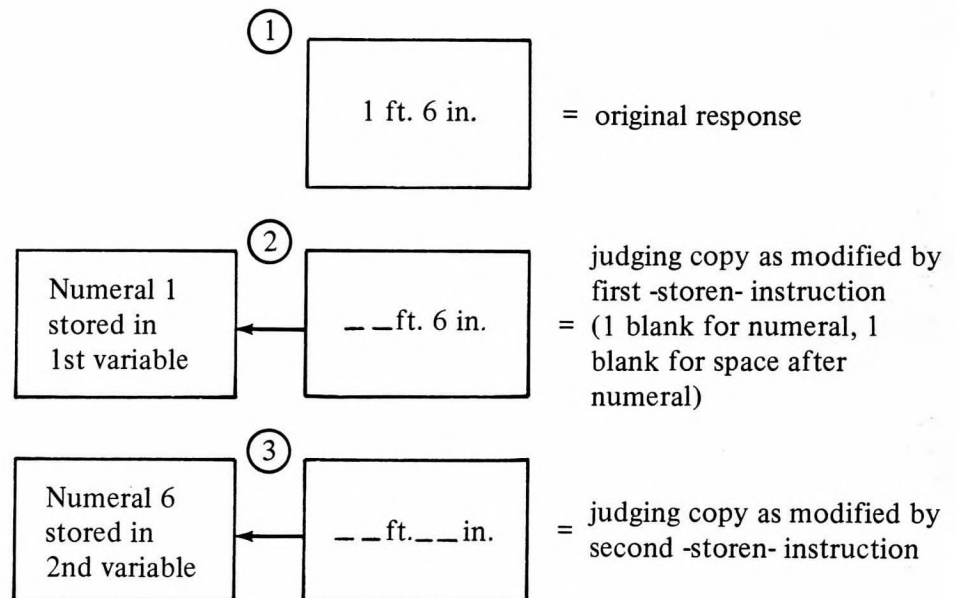


Figure 3. How the -storen- instruction modifies the judging copy

STORAGE CAPABILITIES OF THE -storen- INSTRUCTION

Because the main purpose of a -storen- instruction is to separate a number from a string of characters, certain limitations must be observed. The -storen- instruction accurately stores simple numerics such as integers, or two integers with one operation between them. For example, the -storen- instruction will store:

$3 + 4$
 -7×2
 $2^2 + 1$
 9.5

It also accurately stores decimals such as:

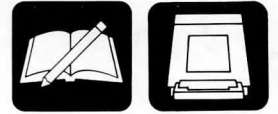
$2.5 + 2.5$
 2.345
 11.34

The storage of complex numerics, however, may yield undesirable results. For example:

- If the response is $3 + (-4)$, the -storen- instruction will store only the 3
- If the response is $2x - 7$, the -storen- instruction will store only the 2
- If the response is $9 \frac{1}{2}$, the -storen- instruction will store 4.5
- If the response is 22.333, the -storen- instruction will store 22.33
- If the response is 3334.7, the -storen- instruction will store 3335

Therefore, do not attempt to substitute the -storen- instruction when the more accurate -store- instruction is required. Use -storen- only when you wish to remove simple numerics from a string of characters in order to use them for calculations.

4-G. APPLICATION PROGRAM



Given programmable-ready material, write a program using instructions presented in this unit.

This activity asks you to write a program using some of the instructions you learned in this unit. Your program should:

1. Request students to type in the months and days of their births. Feedback should be given for unacceptable months and dates, and the students should be required to reenter an appropriate response. (For example, a student might reverse the month and date.)
2. Store the month and date in a way that allows students to type in responses such as:
 - 11th month 30th day
 - 11/30
 - 11-30

If the two numbers are separated by a slash or minus sign, you must be sure that they are interpreted as two numbers.
3. Based on the months and days entered, tell the students their zodiac signs. Table 6 gives the information you will need about zodiac signs.
4. A variable should be set to hold a different value for each zodiac sign, so branching can be accomplished based on the sign. You are not required to code the actual branching.

Do not be surprised or discouraged if you find this lesson difficult to code; it is meant to allow you to experiment, and to assimilate concepts you may not understand as fully as you had thought. If you have difficulty, read the *Hints* section. It provides suggestions for coding this program efficiently, and mentions some of the instructions required. If the *Hints* section does not provide enough information, have no qualms about studying the solution code; it takes time and repeated exposure and practice to be comfortable with the many concepts and instructions presented in *PLATO Author Language: Part II*.

TABLE 6
Information on Zodiac Signs

JAN 20 – FEB 18	Aquarius
FEB 19 – MAR 20	Pisces
MAR 21 – APR 19	Aries
APR 20 – MAY 20	Taurus
MAY 21 – JUN 20	Gemini
JUN 21 – JUL 22	Cancer
JUL 23 – AUG 22	Leo
AUG 23 – SEP 22	Virgo
SEP 23 – OCT 22	Libra
OCT 23 – NOV 21	Scorpio
NOV 22 – DEC 21	Sagittarius
DEC 22 – JAN 19	Capricorn

Note: There are many combinations of instructions that can be used to program this lesson. The hints and solution code point to one combination that the authors feel is acceptably efficient.

Hints

1. A `-putd-` instruction can take care of slashes and minus signs.
2. A `-judge- noquit` can return the system to the first judging instruction if the student enters an inappropriate month number.
3. A `-calcs-` instruction can determine the acceptable number of days for a given month, after which a `-judge- continue` can allow the system to continue in the judging state to accept the second response (the day number).
4. A series of `-calcs-` instructions can be used to assign a value to a variable, depending on the month and day entered. The `-calcs-` can use the logical operations `and` and `or`.
5. The value assigned by the `-calcs-` can be used in a `-writec-` to display the appropriate zodiac sign.

Solution Code

```
define  mth      = v1
        day      = v2
        limit    = v3
        zod      = v4
```

```

unit      Zodiac
at        1010
write    Type in the month and day of your birth.
arrow    1210
putd     ./ .
putd     -. .
storen   mth
write    Use a number for the month.
ok
writec   mth>12 $or$ mth<1*
          The first number should be between 1 and 12.**
judge    mth>12 $or$ mth<1,noquit,x
calcs    mth-2,limit+31,29,31,30,31,30,31,31,30,31,30,31
judge    continue
storen   day
write    Use a number for the day.
ok
writec   day>limit $or$ day<1*
          The second number should be between 1 and (s,limit).
          **
judge    day>limit $or$ day<1,noquit,x
endarrow
calcs    (mth=1$and$day≥20) $or$ (mth=2$and$day≤18),zod+1,,
calcs    (mth=2$and$day≥19) $or$ (mth=3$and$day≤20),zod+2,,
calcs    (mth=3$and$day≥21) $or$ (mth=4$and$day≤19),zod+3,,
calcs    (mth=4$and$day≥20) $or$ (mth=5$and$day≤20),zod+4,,
calcs    (mth=5$and$day≥21) $or$ (mth=6$and$day≤20),zod+5,,
calcs    (mth=6$and$day≥21) $or$ (mth=7$and$day≤22),zod+6,,
calcs    (mth=7$and$day≥23) $or$ (mth=8$and$day≤22),zod+7,,
calcs    (mth=8$and$day≥23) $or$ (mth=9$and$day≤22),zod+8,,
calcs    (mth=9$and$day≥23) $or$ (mth=10$and$day≤22),zod+9,,
calcs    (mth=10$and$day≥23) $or$ (mth=11$and$day≤21),zod+10,,
calcs    (mth=11$and$day≥22) $or$ (mth=12$and$day≤21),zod+11,,
calcs    (mth=12$and$day≥22) $or$ (mth=1$and$day≤19),zod+12,,
at        1510
writec   zod-2*Aquarius*Pisces*Aries*Taurus*Gemini*Cancer*
          Leo*Virgo*Libra*Scorpio*Sagittarius*Capricorn
next     Zodiac

```

5

Pause and Touch Panel Processing

This unit explains instructions that activate the touch panel to allow students to respond by touching the screen. It discusses the use of touch input for pause processing. It also presents programmable-ready material requiring the use of instructions contained in *PLATO Author Language: Part I* and *PLATO Author Language: Part II*.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify the effects of the -enable-, -touch-, -touchw-, disable-, -inhibit- *arrow* and -eraseu- instructions
- Identify solutions to touch panel problems
- Identify the effects of the -pause- instruction with a *keys=touch* tag and the -keytype- instruction with a touch area tag in pause processing
- Given programmable-ready material, write a program using instructions presented in *PLATO Author Language: Part I* and *PLATO Author Language: Part II*

LEARNING ACTIVITIES

-
- 5-A. The Touch Panel. Text/CAI: This activity explains instructions that can be used to allow students to respond by touching the PLATO screen. It also presents practice in the use of touch instructions.

- _____ 5-B. Pause Processing with Touch. Text/CAI: This activity explains the use of touch input in pause processing. It also presents programmable-ready material requiring pause processing with touch input.
- _____ 5-C. A Cumulative Program. Text/Exercise/CAI: This activity presents programmable-ready material for a lesson that requires instructions contained in *PLATO Author Language: Part I* and *PLATO Author Language: Part II*.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subject of pause and touch panel processing, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.

5-A. THE TOUCH PANEL



Identify the effects of the `-enable-`, `-touch-`, `-touchw-`, `-disable-`, `-inhibit-arrow`, and `-eraseu-` instructions.

Identify solutions to touch panel problems.

If you have a touch panel on your PLATO terminal, you've probably used it to move the cursor. The touch panel can also be used to accept student responses; the student can touch a specific area on the screen and be judged correct or incorrect, or be branched to another unit based on the area touched.

This activity discusses the use of the touch panel grid and these touch panel instructions:

- enable- (used to activate the touch panel)
- touch- (used to accept correct responses)
- touchw- (used to accept anticipated wrong responses)
- disable- (used to deactivate the touch panel)

It also discusses system reserve words used with touch instructions, possible problems resulting from the use of the touch panel, and ways to minimize these problems.

THE TOUCH PANEL GRID

The PLATO screen is divided into a grid of 256 touch panel squares or areas. Each square is 32 dots high and 32 dots wide (two lines high and four character widths wide). The lower left-hand corner of each square is always on an even coarse-grid line. Figure 4 illustrates the touch panel grid (reduced size), labeled with coarse-grid markings.

You can see this grid at any time while editing by pressing the TERM key and typing *grid*.

TOUCH PANEL INSTRUCTIONS

Instructions used for touch panel processing include `-enable-`, `-touch-`, `-touchw-`, and `-disable-`.

201		209		217		225		233		241		249		257	
601															
1001															
1401															
1801															
2201															
2601															
3001															

Figure 4. Touch panel grid

The -enable- Instruction

The -enable- instruction makes it possible for the PLATO system to accept touch responses; it activates the touch panel. You must include this instruction before any other touch instructions, and after the -arrow- instruction when using it in arrow processing. Its format is:

```
enable touch
```

The -enable- instruction is in effect until:

- An -endarrow- instruction is encountered
- Another -arrow- instruction is encountered
- The student enters a new main unit
- A -disable- instruction is executed (the -disable- instruction is discussed at the end of this section)

When used in code other than arrow processing code, the -enable- instruction allows students to touch the screen instead of pressing the NEXT key at the end of a unit; it functions as a NEXT key press.

The -touch- and -touchw- Instructions

The -touch- and -touchw- instructions serve the same functions respectively in touch processing as the -answer- and -wrong- instructions do in regular arrow processing; they judge acceptable and anticipated unacceptable responses. These instructions, however, are based on the areas the student touches.

The general formats for -touch- and -touchw- instructions are:

```
touch    AREA1; AREA2; AREA3; . . . . .
```

```
touchw   AREA1; AREA2; AREA3; . . . . .
```

Each area argument specifies a position on the touch panel.

The general format for specifying a touch panel area is: position of lower left-hand corner, characters wide, lines high.

This instruction

```
touch    825, 4, 4
```

specifies an acceptable area (to be judged *ok*) with its lower left-hand corner at coarse-grid position 825. The area is four characters wide, and four lines high (one touch square wide and two touch squares high.)

This instruction

```
touch 1841,8,4
```

specifies an acceptable area with its lower left-hand corner at coarse-grid position 1841. The area is eight characters wide and four lines high (two squares wide and two squares high). The shaded squares in figure 5 show the areas specified by the -touch- instructions just discussed.

This instruction specifies three areas to be judged as anticipated wrong answers:

```
touchw 1817,4,4;2649,8,2;1433,4,8
```

The activated areas are shaded on figure 6.

If you do not specify the width and height of a touch area, the PLATO system assumes you are activating only one square, with its lower left-hand corner at the position specified. For example,

```
touchw 2217
```

activates one square with its lower left-hand corner at position 2217. If any portion of a square is activated, the entire square is activated. Thus, the instruction

```
touch 717
```

activates the entire square, as if the instruction were:

```
touch 817
```

Because the student will not always be touching an area near where the arrow created by the -arrow- instruction is displayed, you may want to prevent the arrow's appearance. To do so, place an -inhibit- *arrow* instruction immediately before the -arrow- instruction. The arrow will not be displayed.

The -disable- Instruction

The -disable- instruction "turns off" the touch panel so that no more touch responses will be accepted. The section in this activity on potential problems with the touch panel shows a legitimate use of the -disable- instruction in arrow processing.

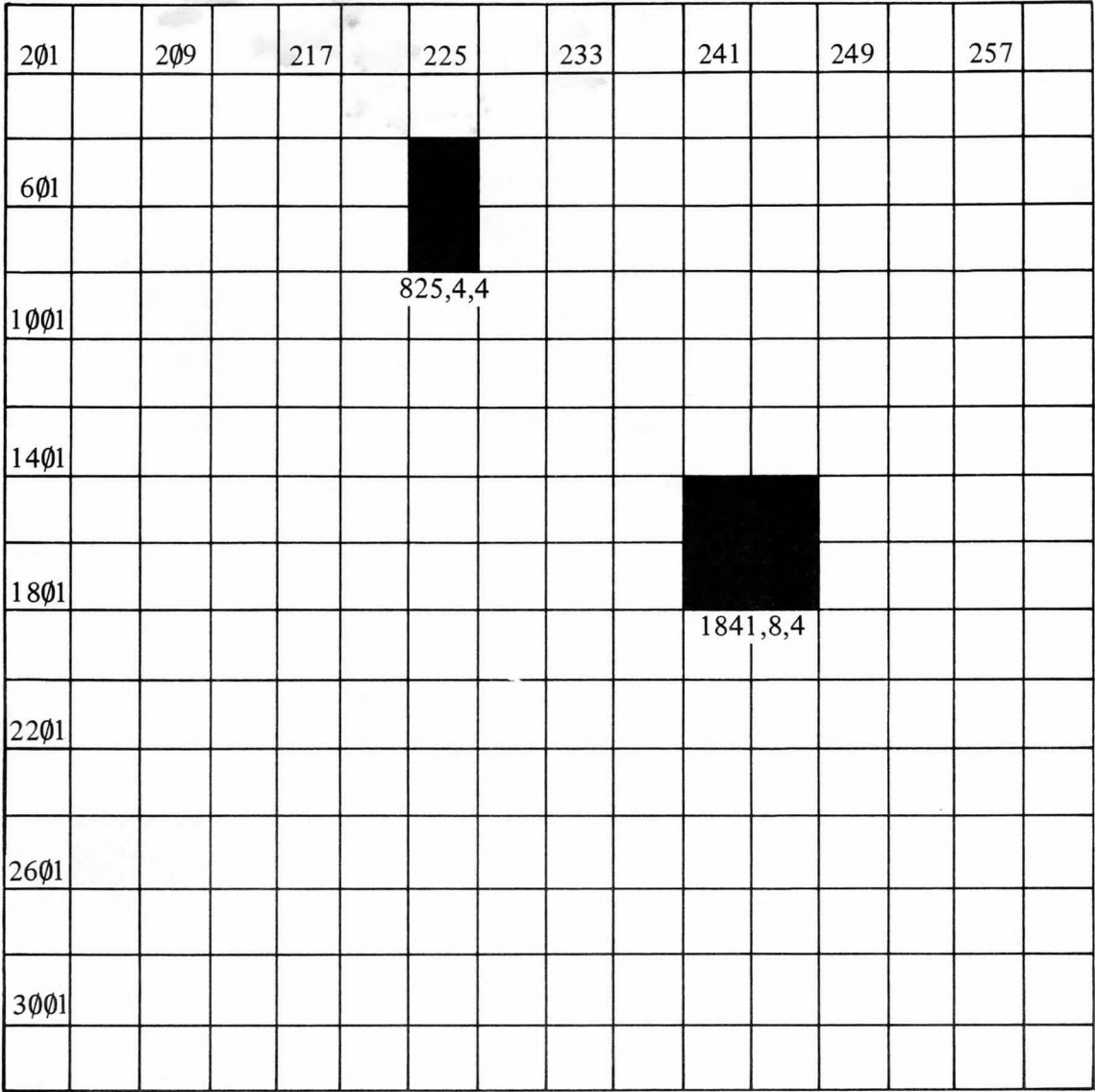


Figure 5. Areas specified by the -ntouch- instructions

201		209		217		225		233		241		249		257	
601															
1001															
1401															
1801															
2201															
2601															
3001															

Figure 6. Activated areas for anticipated wrong answers

SYSTEM RESERVED WORDS

Two system reserved words can be used to record the location touched by a student. The variables *ztouchx* and *ztouchy* hold the fine-grid coordinates of the center of the touched box. If input is other than touch, these variables are set to -1.

POTENTIAL PROBLEMS WITH TOUCH PROCESSING

Student behavior sometimes causes problems with touch panel use. Although touches on the *x* axes are usually accurate, people sometimes slide their fingers up the screen along the *y* axes to the positions they wish to touch. Because the first position touched is the one the PLATO system accepts, it may accept a lower position than that intended by the student.

To minimize this problem, you can write your text on odd-numbered lines. Because the entire square, beginning on an even-numbered line one line down, is activated, you are allowing a margin of one line for "sliding."

It also helps to position touch areas at least four lines apart; this way, acceptable touch positions are separated by a "safety zone"; you can judge as correct any touch falling on the square below the acceptable square.

If you can't separate your acceptable positions, provide visual feedback for the student's response; show the student where the screen was first touched, and allow the response to be corrected.

Another potential problem involves student perception of the area touched; the panel perception may differ. It is helpful to show the area the panel accepted to the student; circle the area, flash the word, or indicate in some visual way which area has been accepted for processing.

A third potential problem involves the fact that the *-enable-* instruction causes touch panel input to act as a NEXT key press. If the student touches the screen more than once when making a response (double touches), the first touch causes the response to be judged; the second touch causes the response to be cleared and the feedback to be erased. The feedback appears to flash on the screen.

There are many solutions to this problem. You can disable the touch panel with the *-disable-* instruction. If you include the *-disable-* instruction after a *-specs-* instruction used as a marker, the touch panel will be disabled after a response has matched a judging instruction. This, however, creates a new problem if you want the touch panel to be active for a new response within the same arrow; after the *-disable-* instruction is executed, no touch responses will be accepted.

If you want to turn the touch panel back on to judge consecutive responses, the *-eraseu-* instruction can be used. In this case, the *-eraseu-* instruction specifies a unit that is to be done before the new response is processed. This unit will be

executed whenever an *ok* or *no* judgment is made and control is returned to the arrow. If the *-enable-* instruction is used in the unit named by the *-eraseu-* instruction, in combination with a *-disable-* instruction after a *-specs-* instruction, you can avoid the frustration caused by double touching.

The following code illustrates the use of the *-eraseu-*, *-disable-*, *-enable-*, and *-specs-* instructions to avoid the double touch problem:

```

unit      touch
:
eraseu    onagain
arrow     1510
enable    touch
specs
disable   touch
touch     .....
:
endarrow

```

```

unit      onagain
enable    touch

```

The first time the arrow is processed, the *-eraseu-* instruction is noted but not executed. The *-enable-* instruction is executed, as would be any regular instruction immediately following an arrow. The PLATO system then waits for a student response. When a response has been entered, the *-specs-* instruction is executed and the PLATO system continues in judging state (the *-disable-* instruction is not yet executed). When a response is matched, the PLATO system returns to the *-specs-* instruction and executes the *-disable-* instruction. It then waits for the student to clear the response. Then the unit named in the *-eraseu-* instruction is executed before a new response can be entered.

Now, however, the student must press the NEXT key whenever a response should be cleared; the student should be informed that a NEXT key press is required for this purpose.

The *-eraseu-* instruction can be used for a variety of other purposes. You may want to look it up in lesson *aids* after completing this activity. You are not required to do so, however.

The two solutions to double touching presented here may not always meet your needs. There are other ways to enable and disable touch. For example, you can incorporate a timing mechanism using the system variable *clock* to enable and disable after a specified period of time. By experimenting, you may find still more ways to avoid the problem of double touching when your program demands yet another solution.

CHECK YOUR UNDERSTANDING**Section 1**

Directions: Write the answers to the questions in the blanks.

1. What instruction activates the touch panel?

2. What instruction judges correct touch responses?

3. What instruction judges anticipated wrong touch responses?

4. Can more than one area be specified in a touch panel instruction?

5. What should you do to see the touch panel grid while editing?

6. What two system variables hold the fine-grid coordinates of the center of the touch panel square the student touched?

7. When the -enable- instruction is used outside of arrow processing, what can it allow the student to do?

8. What four instructions deactivate the touch panel?

9. What instruction prevents the appearance of an arrow when the -arrow-instruction is executed?


Answers

1. -enable- touch
2. -touch-
3. -touchw-
4. Yes
5. Type TERM grid
6. ztouchx and ztouchy
7. Touch the screen instead of pressing the NEXT key at the end of a unit
8. -endarrow-
another -arrow-
-unit- in a new main unit
-disable- touch
9. inhibit arrow


Section 2


Directions: Use the following diagram to answer the questions.

201		209		217		225		233		241		249		257	
601															
1001															
1401															
1801															
2201															
2601															
3001															


1. What instruction accepts touches in the area marked with  as correct responses?



2. What instruction accepts touches on any part of the *b* as correct?

3. What instruction accepts a touch on the  as an anticipated wrong answer?

4. What instruction accepts a touch on any part of the  as an anticipated wrong answer?

5. What instruction accepts a touch on the *a* as an anticipated wrong answer?

6. What one instruction accepts touches on the  and the *b* as alternative correct answers?

7. What one instruction accepts touches on the , , and *a* as anticipated wrong answers?

Answers

1. touch 1017,8,4
2. touch 1813,4,4
3. touchw 1833,8,4
4. touchw 2225,20,2
5. touchw 1045 or touchw 1045,4,2
6. touch 1017,8,4;1813,4,4
7. touchw 1833,8,4;2225,20,2;1045

Section 3

Directions: List three potential problems with the touch panel and identify a solution for each problem.

	<i>Problem</i>	<i>Solution</i>
1.	_____	_____
	_____	_____
	_____	_____
	_____	_____
2.	_____	_____
	_____	_____
	_____	_____
	_____	_____
3.	_____	_____
	_____	_____
	_____	_____
	_____	_____

Answers

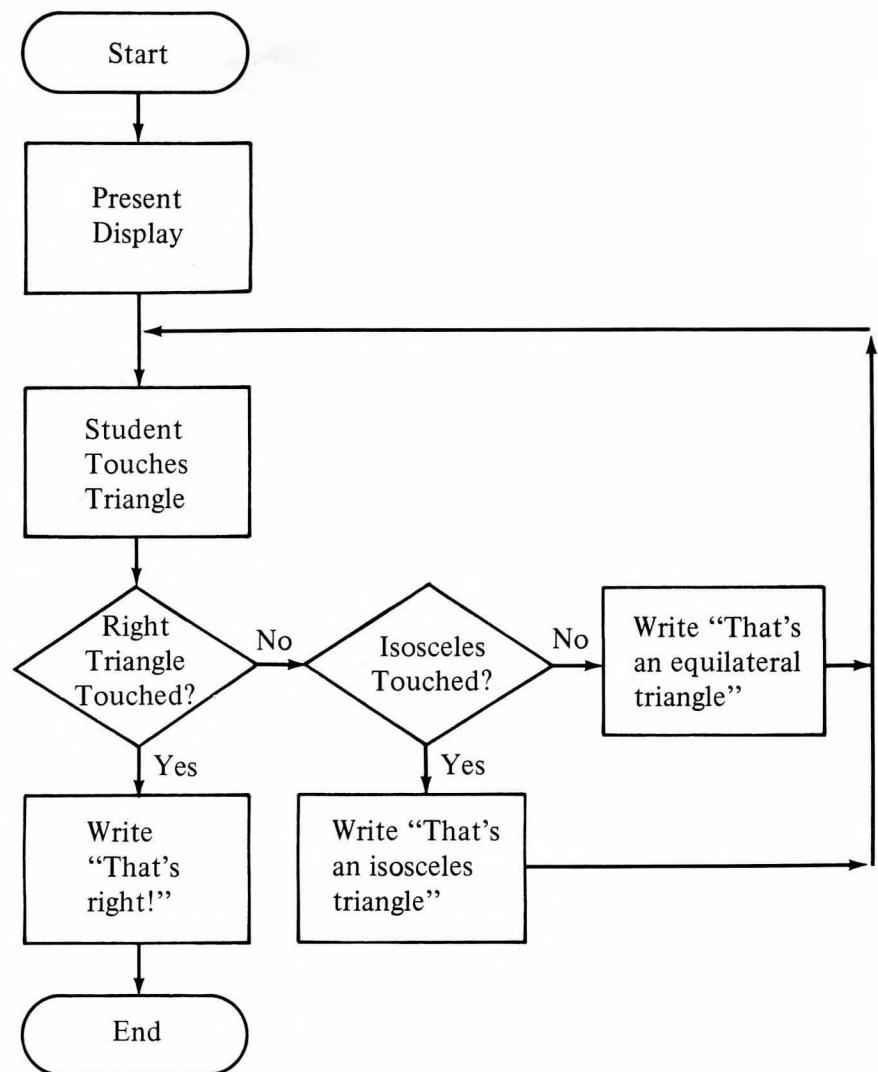
<i>Problem</i>	<i>Solution</i>
1. Sliding fingers up the y axis	a. Write text on odd-numbered lines b. Position touch areas at least four lines apart c. Provide visual feedback for student response
2. Student perception of area touched differs from area accepted by panel	a. Provide visual feedback showing the area accepted by the panel
3. The -enable- instruction acting as a NEXT key press while responses are judged, causing feedback to flash	a. Use the -disable- instruction after a -specs- instruction if only one response is required b. Use the -eraseu- instruction to access a unit containing an -enable- instruction, in combination with a -disable- instruction if more than one response is required c. Use the system variable <i>clock</i> to enable and disable the panel after a specified period of time

Section 4

Directions: Write a program that:

1. Displays a right triangle (a right triangle has one 45-degree angle), an equilateral triangle (all sides are equal), and an isosceles triangle (two sides are equal).
2. Asks the student to touch the right triangle.
3. Gives appropriate feedback (*That's right* if the right triangle is touched, *That's an isosceles triangle* if the isosceles triangle is touched, and *That's an equilateral triangle* if the equilateral triangle is touched). Feedback should be written just below the triangle touched.
4. Prevents the arrow from appearing.
5. Prevents *no* and *ok* judgments from appearing.
6. Prevents flashing feedback from double touches.
7. Ignores unanticipated touches.

A flowchart of the lesson flow follows. A solution to this problem is contained in the *Solution Code* section.



Solution Code

```

unit      triangle
at        510
write     Touch the right triangle.
draw      813;1213;1221;813
draw      1645;1051;1657;1645
draw      2417;1623;2429;2417
inhibit   arrow
eraseu    onagain
arrow     1010
enable    touch
specs     nookno
disable   touch
touch     1213,12,6
    
```

```

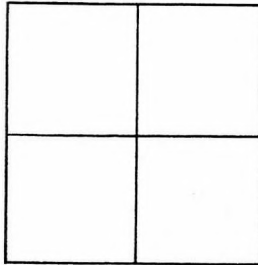
write    That's right!
touchw   2417,12,8
at       2617
write    That's an isosceles triangle.
touchw   1645,12,6
at       1830
write    That's an equilateral triangle.
no
judge    ignore
endarrow
next     triangle
unit     onagain
enable   touch

```

Section 5

Directions: Write a program that:

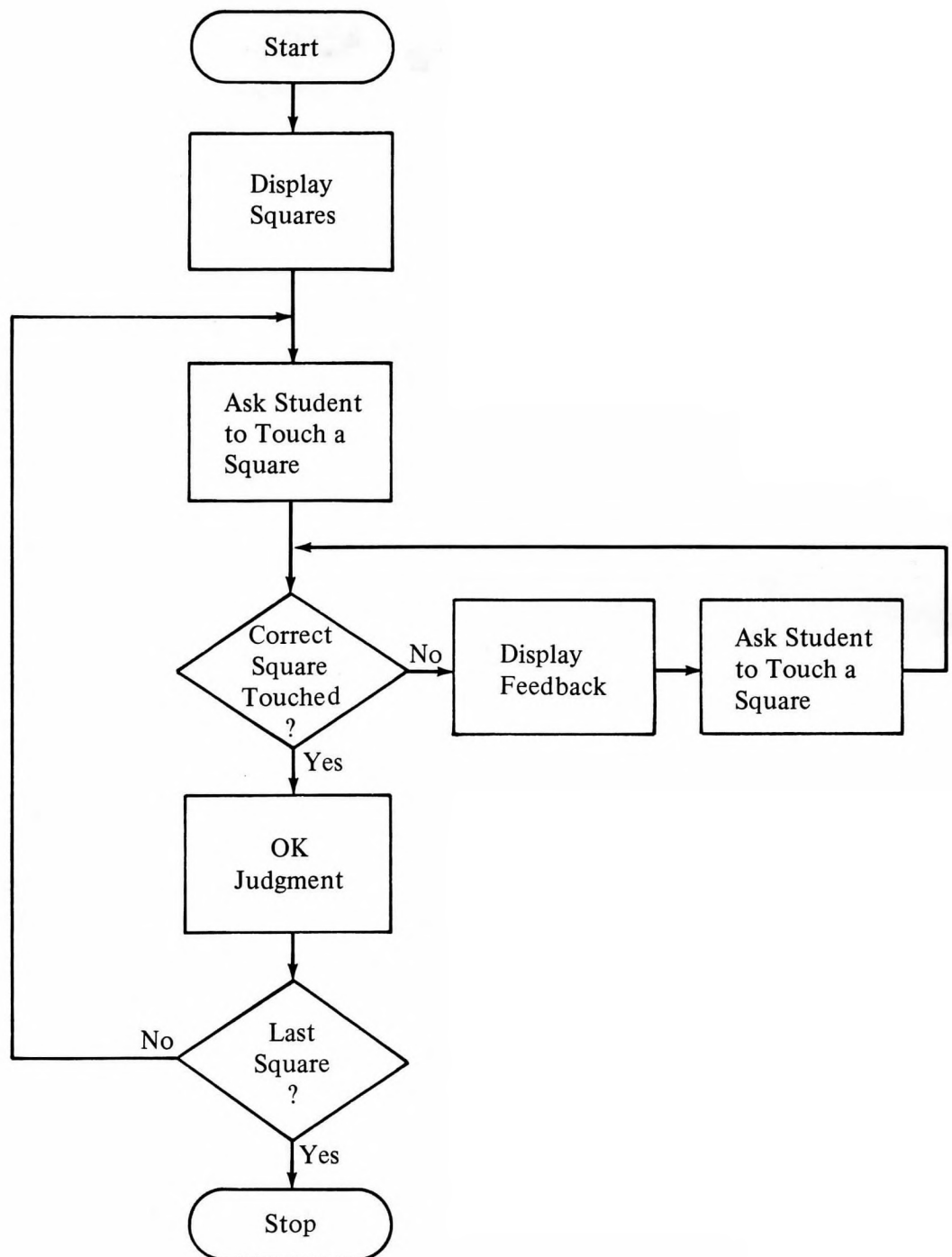
1. Displays a large square divided into four smaller squares:



Each smaller square should be eight characters wide and four lines high.

2. Asks students to touch squares in this order:
 - a. Upper left-hand corner
 - b. Upper right-hand corner
 - c. Lower left-hand corner
 - d. Lower right-hand corner
3. Judges acceptable responses *ok*.
4. Judges anticipated wrong answers with *no* and feedback for the square touched, such as *You touched the _____ corner.*
5. Ignores unanticipated touches.
6. Does *not* display the arrow.
7. Prevents flashing feedback.
8. Ends the lesson after the final square is touched and judged *ok*.

A flowchart of the lesson flow follows.



This lesson is more difficult to program than the lesson in section 4. Suggestions for efficient coding are included in the *Hints* section. A documented solution is included in the *Solution Code* section.

Hints

You may want to follow these suggestions when coding your lesson.

1. A -doto- loop can be used to: (1) present the four requests; and (2) judge all responses to all requests.
2. A conditional -join- instruction can be used to pass the location of the lower left-hand corner for all correct responses; thus, correct responses can be judged in a compact auxiliary unit.
3. Anticipated incorrect locations can be judged for *all* requests by including four -touchw- instructions after the -join- instruction (all four anticipated touch locations). The correct response will be listed among the anticipated incorrect responses; this is acceptable, because the -touch- instruction is executed first.

Solution Code

```

define square = v1
      wk1      = v2
unit    squares
doto    1loop,square←-1,2  $$ value used to display
*                               appropriate request.
at      1010
writec  square←Touch the upper left-hand corner of the box.←
      Touch the upper right-hand corner of the box.←
      Touch the lower left-hand corner of the box.←
      Touch the lower right-hand corner of the box.←
draw    2025;1225;1241;2041;2033;1233;skip;1625;1641
*                               ↑ draws squares
eraseu  onagain  $$ prevents flashing feedback
inhibit arrow
arrow   2010
enable  touch
specs
disable touch
join    square,rt(1625),rt(1633),rt(2025),rt(2033)
*  ↑ passes arguments to unit rt, based on the value of
*    square. Arguments are correct touch locations for
*    each request.
touchw  1625,8,4
write   You touched the upper left-hand square.
touchw  1633,8,4
write   You touched the upper right-hand square.
touchw  2025,8,4

```

```

write  You touched the lower left-hand square.
touchw 2033,8,4
write  You touched the lower right-hand square.
*  ↑ Each of the four squares is identified as a
*  wrong touch in one of the four -touchw-
*  instructions above. However, these instructions
*  will not be executed if a match is found in the
*  joined unit, rt(wk1).
no
judge  ignore  $$ ignores unanticipated responses
endarrow
if      square<2  $$erases feedback
.      pause    keys=next,term
.      at       1010
.      erase    45
.      at       2013
.      erase    2
endif
1loop
end      lesson
***
unit    onagain  $$ this unit is used to activate the
*                               panel after an incorrect response
enable  touch
***
unit    rt(wk1)  $$ left-hand corner location is passed
*                               to wk1
touch   wk1,8,4  $$ correct responses for all requests
*                               judged here

```

5-B. PAUSE PROCESSING WITH TOUCH



Identify the effects of the `-pause-` instruction with a *keys=touch* tag and the `-keytype-` instruction with a touch area tag in pause processing.

PLATO Author Language: Part I explains the fundamentals of pause processing, in which the `-keytype-` and/or `-keylist-` instructions are used in conjunction with the `-pause-` instruction. You may remember this sample code in which the instructions branch the student depending on the value of the variable in the `-keytype-` instruction tag:

```
define  press    = v1
unit    index
at      510
write   Press a, b, or c to indicate which part of the
        activity you would like to see:
        a.  Introduction
        b.  Examples
        c.  Quiz

1start
pause   keys=all
keytype press,a,b,c
branch  press,1start,x
jump    press-1,intro,example,quiz
***
unit    intro
:
:
unit    example
:
:
unit    quiz
:
:
```

With pause processing, you control the branching when the student responds. This differs from arrow processing, in which the system controls much of the branching, such as the branch back to the `-arrow-` instruction when the student response matches a `-wrong-` instruction. Pause processing is particularly useful with touch input. This activity discusses the use of the `-pause-` and `-keytype-` instructions in conjunction with touch input.

PAUSE PROCESSING WITH TOUCH INPUT

The `-pause-` instruction can activate the touch panel and allow a touch to break the pause simultaneously. No `-enable-` instruction is needed. The format is:

```
pause    keys=touch
```

This instruction can be used in conjunction with a `-keytype-` or `-keylist-` instruction to branch a student depending on the area the student touches. The following instructions produce these results:

- These words are displayed on the screen:

Touch spot 1

Touch spot 2

Touch spot 3

Touch spot 4

Touch spot 5

Touch spot 6

Touch spot 7

Touch spot 8

Touch spot 9

- Students can choose a spot by touching.
- If a spot not listed in the `-keytype-` instruction is pressed, the student returns to the `-pause-` instruction so an appropriate spot can be pressed.
- When an appropriate spot is pressed, the student “falls through” and the `-show-` instruction displays the value held in variable *spot*.

Note the format for including touch areas in a `-keytype-` instruction (the letter *t*, the parentheses, and the comma after the left-hand corner specification):

```
define  index    = n1
        spot     = n2
unit    test
doto    1write, index+0, 8
at      909+index*200
write   Touch spot (s, index+1)
1write
```

```

1pause
pause    keys=touch,term
keytype  spot,t(1009,11,2),t(1209,11,2),
         t(1409,11,2),t(1609,11,2),
         t(1809,11,2),t(2009,11,2),
         t(2209,11,2),t(2409,11,2),
         t(2609,11,2)
branch   spot,1pause,x
at       3203
show     spot

```

To box the spot touched, this instruction could be added following the -branch- instruction:

```

box      908+200*spot;822+200*spot

```

CHECK YOUR UNDERSTANDING

Section 1

Directions: Write a program that:

1. Displays these four choices:
 - a. Metric Units of Length
 - b. Metric Units of Temperature
 - c. Metric Units of Volume
 - d. Metric Units of Weight
2. Allows the student to choose a selection by touching the selection.
3. Allows for branching based on the chosen selection. You do not have to program the units to which the students will branch. Merely insert a -unit- instruction for each unit, and -at- and -write- instructions showing you which unit you have branched to. For example, *We're in unit weight.*

A solution for this problem is contained in the *Solution Code* section.

Solution Code

```

define   spot=n1
unit     choice
at       515
write    Touch the section you would like to
         take at this time.

```

a. Metric Units of Length

b. Metric Units of Temperature

c. Metric Units of Volume

d. Metric Units of Weight

```

1pause
pause    keys=touch,term
keytype  spot,t(1017,28,2),t(1417,32,2),t(1817,28,2),
          t(2217,28,2)
branch   spot,1pause,x
jump     spot-1,length,temp,vol,weight
unit     length
at       1010
write    We're in unit length.
next     choice
unit     temp
at       1010
write    We're in unit temp.
next     choice
unit     vol
at       1010
write    We're in unit vol.
next     choice
unit     weight
at       1010
write    We're in unit weight.
next     choice

```

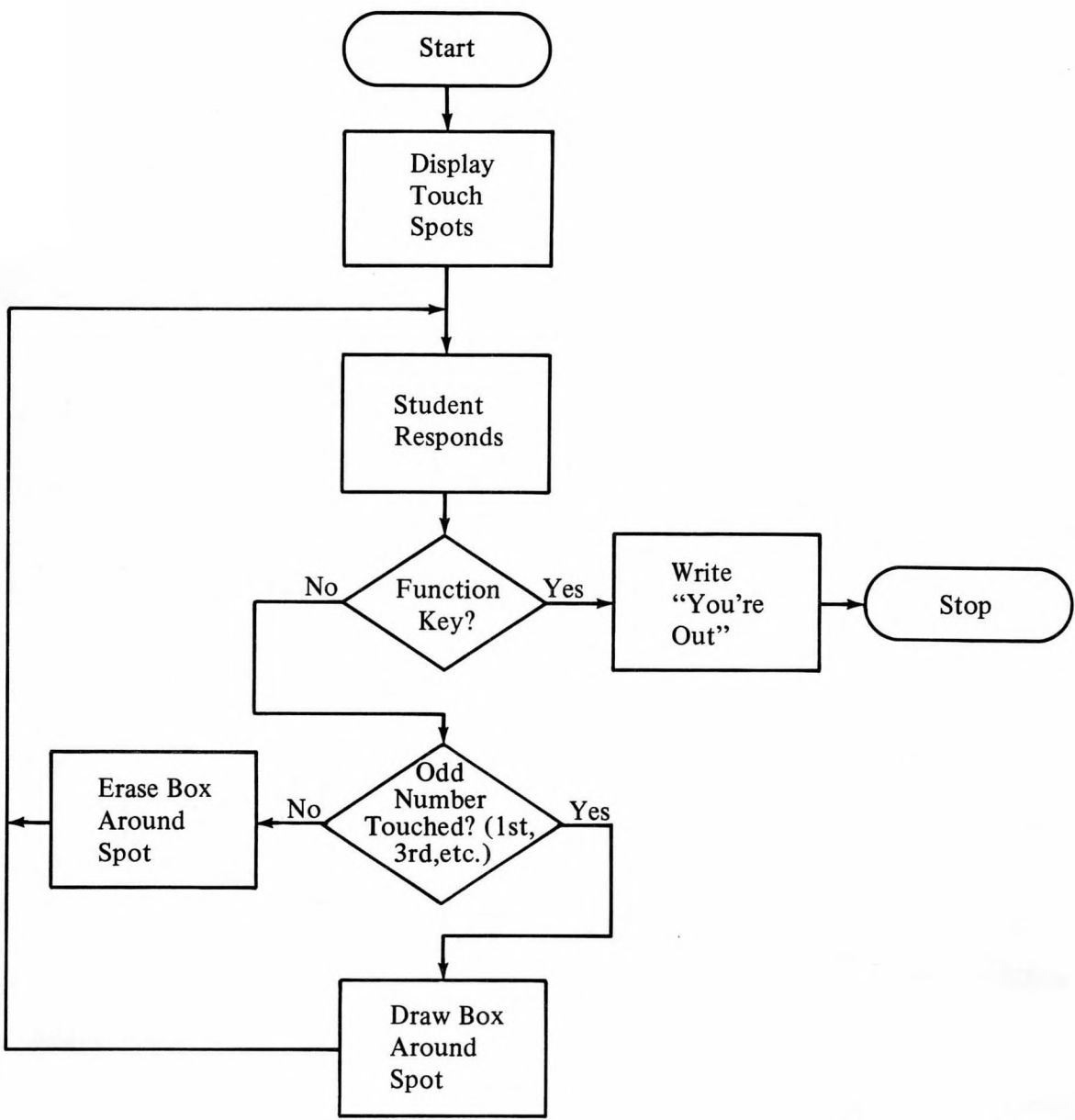
Section 2

Directions: In this exercise, you will expand on the example presented in the text. Your program should:

- Display nine touch spots.
- Box the spot touched by the student.

- Allow students to touch as many spots as they wish.
- Allow students to change their minds about spots already touched; if a spot is touched a second time, the box should be erased; if the spot is touched a third time, the box should appear again, and so on.

When the student has finished choosing spots to be boxed, a function key should be available to end the loop. The student should be told that the loop has ended. A flowchart of this process follows.



Although this exercise serves no educational purpose for your students, it shows you how pause processing with touch can allow students to make reversible decisions (on an index, for example).

This lesson is more difficult to program than the lesson in section 1. If you have problems coding this lesson, you may wish to read the hints in the next section of this activity. Two solutions for the lesson are included after the *Hints* section.

Hints

To code this lesson, you might want to consider the following suggestions:

1. Use nine consecutive variables defined so that they may be accessed by an index, one to represent the status of each touch spot. Because the status is binary (on, off), you may use a variable that has been segmented by ones. The values of these nine variables (or the nine bits of a segmented variable) can be used to represent the present status; for example, the value 1 can represent on, and the value 0 can represent off.
2. Use a -calcs- to change the status for each box as it is touched; change the status value to 0 if it was 1, and change it to 1 if it was 0.
3. Use a conditional -mode- to draw the box around the spot in either mode write or mode erase, depending upon the value of the status variable for that particular spot.
4. Use -pause-, -keytype-, and -branch- instructions as they have been used in the example in the text.

Solution Codes

```
1. define  loop      = n1
          keypres    = n2
          segment,tch=n3,1
          stch       = n3

***
zero      stch
***
unit      test
doto      1write,loop+0,8
at         909+loop*200
write     Touch spot <(s,loop+1)>
1write
at         3235
write     LAB when you're finished
1pause
```

```

pause    keys=touch, funct
keytype  keypres, t(1009, 11, 2), t(1209, 11, 2),
          t(1409, 11, 2), t(1609, 11, 2), t(1809, 11, 2), t(2009, 11, 2)
          t(2209, 11, 2), t(2409, 11, 2), t(2609, 11, 2), lab
branch   keypres, 1pause, x, x, x, x, x, x, x, x, 2out
mode     tch(keypres+1)=0, write, erase
box      908+200*keypres; 822+200*keypres; 2
calcs    tch(keypres+1)=0, tch(keypres+1)≠1, 0
branch   1pause
2out
mode     write
at       3203
write    We're out.

```

```

2. define loop    = n1
           keypres = n2
           segment, tch=n3, 1
           stch    = n3

***
zero    stch
***
unit    test
doto    1write, loop≠0, 8
at      909+loop*200
write   Touch spot <s, loop+1>
1write
at      3235
write   LAB when you're finished
1pause
pause   keys=touch, funct
keytype keypres, lab, t(1009, 11, 2), t(1209, 11, 2),
          t(1409, 11, 2), t(1609, 11, 2), t(1809, 11, 2), t(2009, 11, 2)
          t(2209, 11, 2), t(2409, 11, 2), t(2609, 11, 2)
branch  keypres, 1pause, 2out, x
mode    tch(keypres)=0, write, erase
box      708+200*keypres; 622+200*keypres; 2
calcs    tch(keypres)=0, tch(keypres)≠1, 0
branch   1pause
2out
mode     write
at       3203
write    We're out.

```

SUMMARY

This instruction

`pause keys=touch`

can be used in conjunction with the `-keytype-` instruction in pause processing. It activates the touch panel and allows a touch to break the pause.

The `-keytype-` instruction branches the student based on the area touched. Its format is:

`keytype VARIABLE,t (CORNER,CHARACTERS WIDE,LINES HIGH)`

5-C. A CUMULATIVE PROGRAM



Given programmable-ready material, write a program using instructions presented in *PLATO Author Language: Part I* and *PLATO Author Language: Part II*.

Now that you are nearing the end of the course, it is appropriate that you create a program that requires the use of many of the instructions you have learned.

One of the challenges of programming is to code a lesson in the most efficient way possible with the instructions at your disposal. For this lesson, you should mentally sort through the instructions currently at your command, and choose those most appropriate for your purposes.

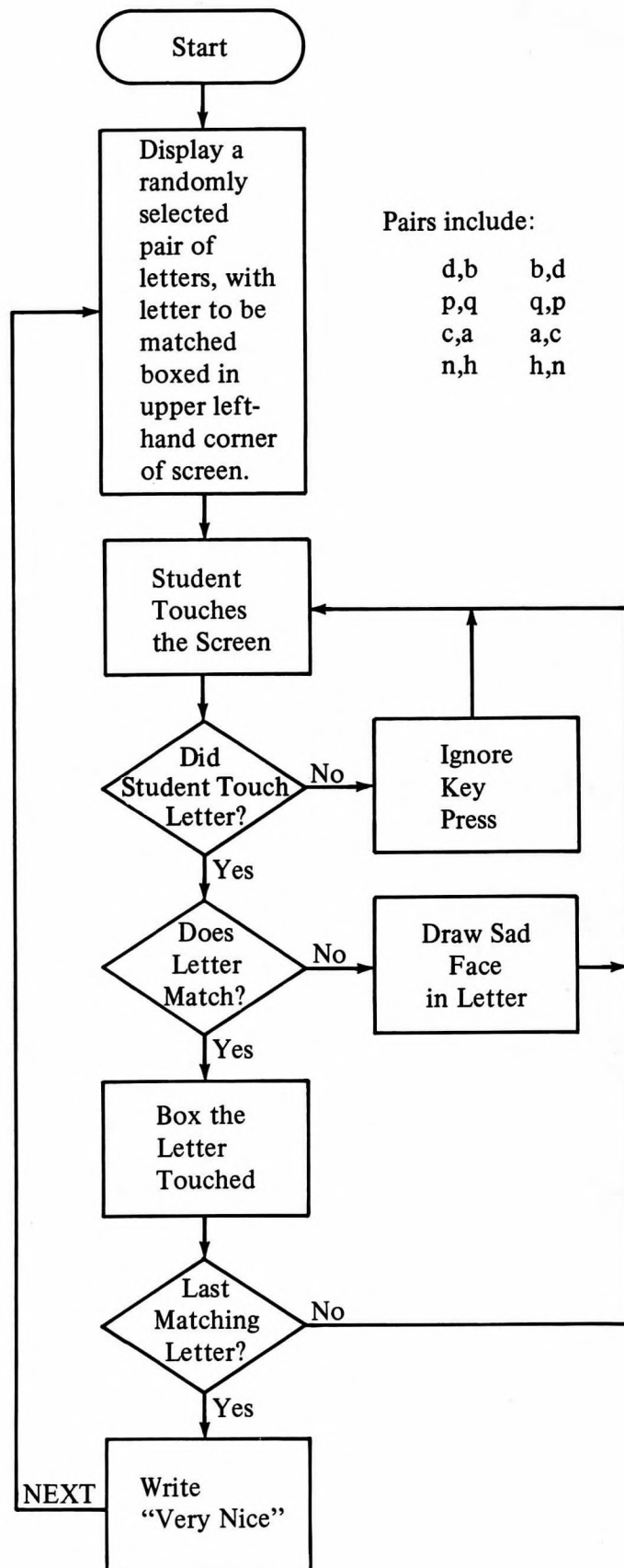
Directions: The following pages contain the goal statement, flowchart, and story-book for this lesson. Read these materials and code a lesson producing the desired results; it may not be as easy to code as it appears.

If you have problems, the section following the programmable-ready material can help you. It contains acceptable code for the entire lesson, with comments included to help you understand the code. You may wish to study the code, and then put the code away and try to code it yourself. Then, if you encounter more problems, study the next section of code and attempt to program this section yourself. If you proceed in this way, you will come to understand the rationale for each section of the solution code.

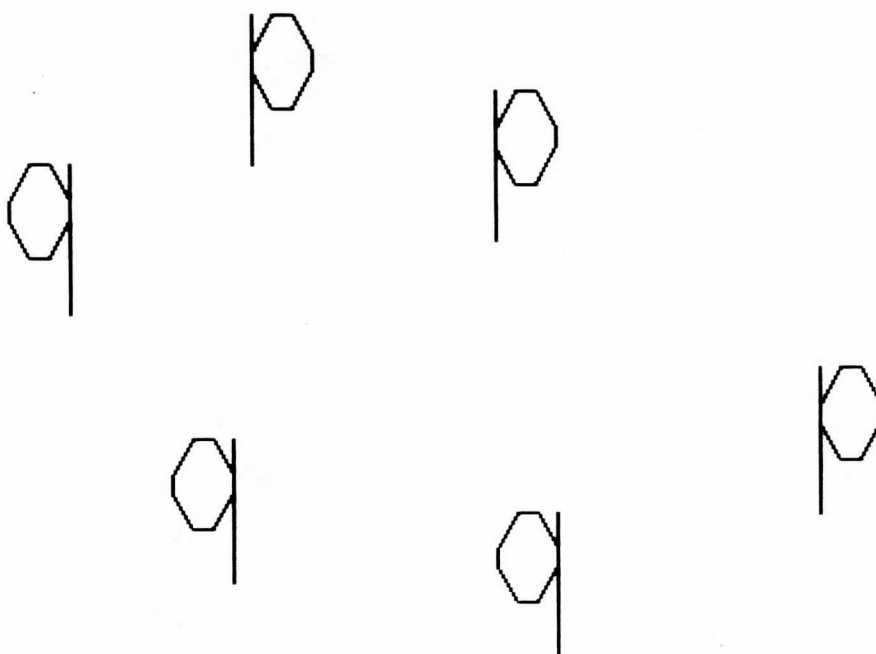
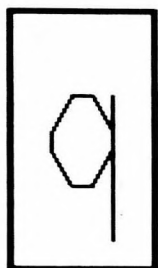
GOAL STATEMENT

This lesson is a visual discrimination drill for beginning readers. The students are presented with pairs of similar letters. They must touch the letters that match the boxed letter in the upper left-hand corner of the screen.

Pairs of letters should be selected randomly. The lesson should be an endless loop; the students' instructors will sign them off the terminal when they see fit.



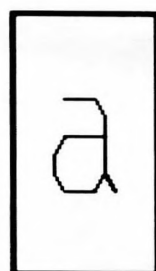
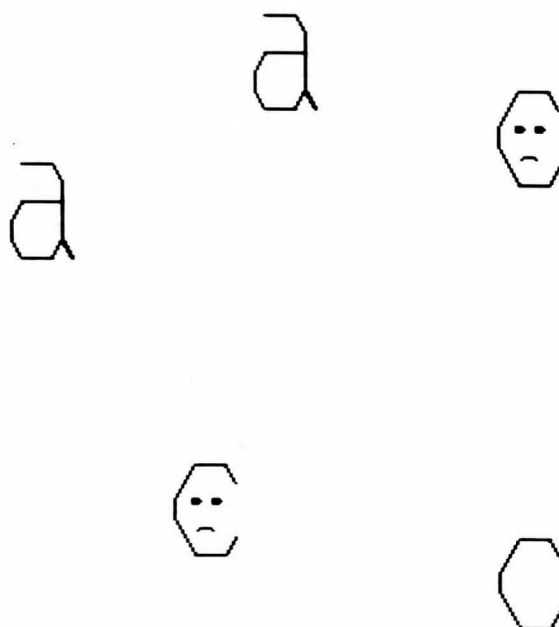
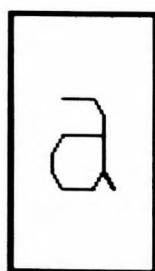
STORYBOOK



The following pairs of letters should be presented in random order:

d,b	b,d
p,q	q,p
c,a	a,c
n,h	n,h

Three of each letter should be displayed. The letters should be sized. The display above shows the desired size, placement, and format of the letters.



When students touch a matching letter, the letter should be boxed to look just like the letter in the left-hand corner. If students touch unmatched letters, a sad face should be drawn within the letter. All other touches should be ignored.

When all matching letters have been touched, the student is shown a new display of letter pairs.

Solution Code

```
*d,b;p,q;c,a;;n,h
define wk1    = n1    $$ temporary variables that can be
   wk2        = n2    $$ written over as program progresses
   loc        = v3
   hap        = v4
   org        = v5
```

```

let1      = n6  $$ letter 1 of pair
let2      = n7  $$ letter 2 of pair
segment,doneit=n8,1
wdoneit = n8
place(i) = n(8+i)  $$ requires n9 thru n14.  a
*                                one-dimensional array.
***
unit      set          $$ to select letters and set placement
zero      wdoneit      $$ zeros all segments of doneit
setperm 6          $$ sets upper limit for random selection
doto      1rand,wk1≠1,6
randp     wk2          $$ randomly selects a number 1-6 without
*                                replacement
calcs     wk2-2,place(wk1)≠1225,1437,1613,2253,2421,2637
*                                † places these six coarse grid positions into
*                                six sequential variables.  Positions are
*                                randomly selected by the random selection
*                                of wk2.  Because this loop is repeated six
*                                times and we are using a -setperm- -randp-
*                                combination, all of these coarse grid
*                                positions will eventually be selected.
1rand
randu     wk1,2  $$ randomly selects which of the two letters
*                                of the pair will be the correct answer
randu     wk2,4  $$ randomly selects the set of pairs for
*                                the problem
if        wk1=1  $$ † these values are assigned if wk1=1
.         calcs  wk2-2,let1≠'d','p','c','n'
*                                † one of these character strings is placed
*                                in let1 depending upon the value in wk2
.         calcs  wk2-2,let2≠'b','q','a','h'
*                                † its partner is placed in let2 depending
*                                upon the value in wk2
else      $$if wk1=2, the character strings are selected
*                                opposite from what would have been selected if
*                                wk1=1
.         calcs  wk2-2,let2≠'d','p','c','n'
.         calcs  wk2-2,let1≠'b','q','a','h'
endif
jump      touch
*
unit      touch
back      set
box       805;112;2  $$ box for upper left-hand corner

```

```

size      4,8      $$ size for letters
at        607      $$ position of letter in left-hand corner
showa     let1      $$ show boxed letter to be matched as
*                               held in let1
doto      1write,wk2+1,6  $$ begins at value of 1, ends at
*                               value of 6; displays 6 letters
do        wk2>3,let(let2,place(wk2)),let(let1,place(wk2))
*                               + passing arguments. If wk2>3, unit let is
*                               done and wk4 in that unit is given the value
*                               of let2, while loc is given the value of
*                               place(4), place(5), or place(6). Otherwise,
*                               the opposite letter of the pair is placed
*                               at place(1), place(2), or place(3).
1write
size      0        $$ size is back to normal
inhibit   arrow     $$ no arrow is displayed
doto      3get,wk2+1,3  $$ three times because there are
*                               three correct letters
arrow     where+2
enable    touch     $$ touch is activated
specs     nookno     $$ no ok or no is displayed on the screen
join      goodt(place(1))  $$ unit that judges ok and gives
*                               feedback for correct responses.
*                               The passed argument is a correct
*                               touch location.
join      goodt(place(2))
join      goodt(place(3))
join      badt(place(4))  $$ unit that judges no and gives
*                               feedback for incorrect touches.
join      badt(place(5))
join      badt(place(6))
endarrow
3get
at        3220
write     very nice!
next      set
*
*
unit      face(hap,org)  $$$sad face if hap=0, else happy
if        hap=0  $$ draw sad face if hap = 0; otherwise,
*                               box the letter
.         rorigin org
.         rat      8,28
.         rcircle 2  $$ one eye

```

```

.      rat      16,28
.      rcircle 2  $$ another eye
.      rat      12,12
.      rcircle 5,143,37  $$ mouth
else
.      box      place(anscnt)+198;place(anscnt)-495;2
*              † box placed around matched letter
endif
***
unit    let(wk1,loc)  $$ a "done" unit
at      loc  $$ the value of place(wk2) has been passed to
*              this variable when this unit was "done"
showa   wk1  $$ either the character string in variable let1
*              or let2 has been passed to this variable
*              when this unit was "done"
***
unit    goodt(org)  $$ this unit is "joined" three times.
*              the value of place(1), place(2) or
*              place (3) is passed to org
touch   org,4,4  $$ org sets left-hand corner. Area is
*              4 characters wide, 4 lines high
judge   doneit(anscnt)=1,ignore,x
calc    doneit(anscnt)≠1
do      face(-1,org)  $$ this value causes a box to be
*              drawn around the matched letter
***
unit    badt(org)  $$ value in place(4), place(5), or
*              place(6) placed in org
touchw  org,4,4  $$ sets left-hand corner, 4 characters
*              wide, 4 lines high
do      face(0,org)  $$ passes arguments to unit face, to
*              what will be drawn and where it will
*              be drawn
judge   ignore  $$ eliminates the need for key or touch
*              input to clear the incorrect response

```

6

Lesson Control

Though the information presented in this unit is not of immediate use to you because of the limited number of lessons you have programmed, it will be of value in the future. This unit discusses the use of the -jumpout- instruction to connect lessons. It also discusses the routers available on the PLATO system; routers control the student flow through the lessons of a course.

INSTRUCTIONAL OBJECTIVES

After completing this unit, you should be able to:

- Identify the effects of the -jumpout- instruction
- Identify characteristics of PLATO routers

LEARNING ACTIVITIES

- _____ 6-A. Connecting Lessons with the -jumpout- Instruction. Text/Exercise: This activity explains forms of the -jumpout- instruction. This instruction is used to “jump” students from one lesson to another and back again.
- _____ 6-B. Routers. Text/Exercise: This activity discusses the purpose of routers and explains characteristics of available routers.

TESTING DIRECTIONS

Now sign on to a Control Data PLATO terminal. If you are very familiar with the subjects of the -jumpout- instruction and routers, you may wish to test out of some of the activities in this unit. If you are not familiar with this topic, do not take the test. Instead, when the module description page appears, press the letter *a* for a complete list of assignments for this unit.



6-A. CONNECTING LESSONS WITH THE -jumpout- INSTRUCTION

Identify the effects of the -jumpout- instruction.

When you have coded a number of lessons, you may find that one lesson, or a portion of a lesson, is appropriate for use in another lesson. The -jumpout- instruction allows you to connect these lessons so students can be jumped back and forth between compatible lessons.

The -jumpout- instruction can be used with a variety of tags. This activity discusses four commonly used tags. Other tags are listed in lesson *aids*; you need not study the tags in *aids* for this course, but you may wish to look them up for your own information.

TWO BASIC TAGS

The formats for the two most basic -jumpout- instruction tags are:

```
jumpout LESSON NAME  
jumpout LESSON NAME, UNIT NAME
```

With the first format, the student jumps from the current lesson to the beginning of the lesson named in the -jumpout- instruction. This instruction

```
jumpout trial
```

jumps the student to the beginning of lesson *trial*.

The second format jumps the student from the current lesson to a specific unit of another lesson. This instruction

```
jumpout trial,error
```

jumps the student to unit *error* in lesson *trial*.

With both formats, the IEU is executed before the students begin the lesson to which they have jumped.

THE CONDITIONAL TAG

The -jumpout- instruction can also be used conditionally. The general format is:

```
jumpout VARIABLE; LESSON; LESSON; LESSON
```

or

```
jumpout VARIABLE; LESSON, UNIT; LESSON, UNIT; LESSON, UNIT
```

The student is branched to alternative lessons, depending on the value of the variable. The first alternative is performed if the value is -1, the second if the value is 0, and so forth.

RETURNING TO THE ORIGINAL LESSON

Two -jumpout- instruction tags are used to return the student to the original lesson upon completion of the specified section of the lesson to which the student was jumped. The formats are:

```
jumpout return
jumpout return, return
```

The first format returns the student to the beginning of the original lesson. This might be used when the beginning of the original lesson contains an index.

The second format also returns the student to the original lesson; however, the student is returned to the unit following the unit containing the -jumpout- instruction. Thus, if you want the student to return immediately to the unit containing the -jumpout- instruction (to an index, for example) you must include a "dummy" unit; this dummy unit exists solely for the purpose of branching the student back one physical unit, as shown in figure 7.

Caution: The -jumpout- instruction should be used with these factors in mind:

- The accessed lesson must be condensed, which requires a considerable amount of processing time.
- If not enough ECS space is available when the -jumpout- instruction is executed, the student will be "dumped"; an error message is displayed.
- If the jumpout is to a specific unit of a lesson, the jumpout code of the lesson jumped to must match the jumpout code of the lesson jumped from.

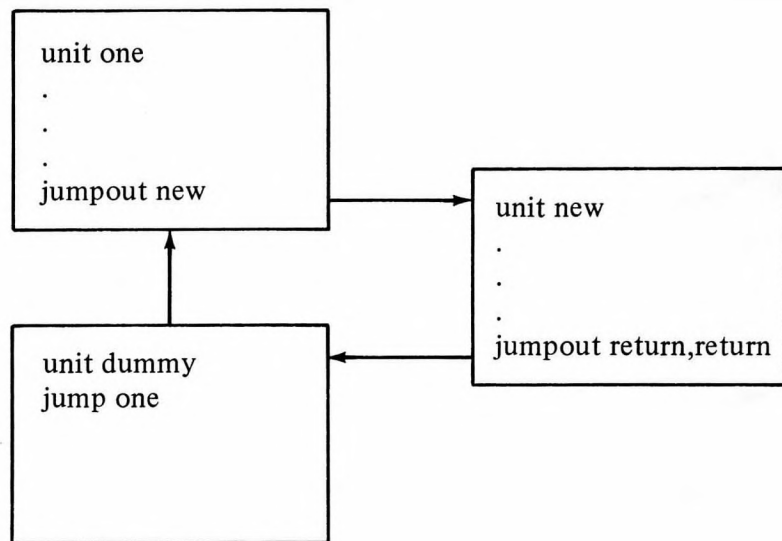


Figure 7. Using the dummy unit

CHECK YOUR UNDERSTANDING

Directions: Answer the questions by filling in the blanks.

1. Which form of the -jumpout- instruction jumps the student from the current unit to the beginning of the lesson named in the -jumpout- instruction?

2. Which form of the -jumpout- instruction jumps the student from the current lesson to a specific unit of another lesson?

3. Is the IEU executed with both of the formats asked for in questions 1 and 2?

4. What are the conditional formats of the -jumpout- instructions you listed in questions 1 and 2?

5. Which form of the -jumpout- instruction returns the student to the beginning of the lesson in which the previous -jumpout- was executed?

6. Which form of the -jumpout- instruction returns the student to the unit following the unit containing the previous -jumpout- instruction?

7. What can you do to return the students to the unit from which they jumped?

Answers

1. jumpout lesson name
2. jumpout lesson name,unit name
3. Yes
4. jumpout variable;lesson;lesson;lesson
jumpout variable;lesson,unit;lesson,unit;lesson,unit
5. jumpout return
6. jumpout return,return
7. Include a “dummy” unit to branch the student back one unit



6-B. ROUTERS

Identify characteristics of PLATO routers.

At some time in the future, when you have programmed a series of lessons, you may be called upon to set up a router to organize the lessons; a router directs the students in a course through a predetermined sequence of lessons. When they complete a lesson, students are routed to the next lesson in the sequence.

This activity describes characteristics of two types of routers. The first is the system router, called the *mrouter*. The second is the router that you, the author, can create. Descriptions of both types of routers are brief; they are meant merely to alert you to the existence and basic capabilities of these routers.

THE SYSTEM ROUTER

The system router was created and is maintained by the PLATO system staff. Its advantages are:

1. The basic router is already set up for you so you don't have to spend time programming the router itself.
2. Only fifty words of ECS are charged for use of this router. With an author-created router, you are charged ECS for the entire router at all times, even when a lesson is being executed.

Components of the System Router

In order to use the system router, you must ask your account director to create a *group* and an *instructor file* for you. The group is where you register your students. You must tell the account director how many parts you want in your group; one part can carry six students. You will be allowed to name your group.

The instructor file personalizes the *mrouter*; it enables the system to distinguish between your course and the courses of other authors. Your instructor file will also have a name and a specified number of parts.

The space in an instructor file is divided between the two components of an instructor file, the *catalog of lessons* and the *module space*. A trade-off in space is involved; the more catalog space you use, the less module space you have.

The catalog of lessons is a list of all the lessons in your course. From this catalog, you have selected lessons to form modules; different students can then be assigned different modules. Module space is required to display an index of lessons to the students.

Setting Up the Router

Once you have been assigned a group, you can then set change and inspect codes for your group, and write a description of the group. The change and inspect codes prevent other authors from editing your group. On the information display, you will be asked to choose a router, and you should type in *mrouter*. When *mrouter* is chosen, another option is displayed; this option is not displayed for other routers. It asks for your instructor file name, which must be typed in.

To set up your instructor file, you must sign on, typing your group name on the author mode display. The first time you access your instructor file, you will be asked to specify change and use codes; these codes prevent others from attaching and changing your instructor file. You may want to tell others your *use* code, to allow them to attach and *use* your instructor file. You will then see an index containing a *Curriculum Design* option. After choosing this option, you will see another index. This index includes an option that allows you to access your catalog space. You then add lessons to your catalog, following the instructions on the displays.

After you have put lessons in your catalog, you can design your modules. You must decide which lesson in your catalog will be included in each module. When the students begin your course, they will see a module index display; this display lists the lessons the student has been assigned. You can change the module index page to make it more attractive, if you wish.

AUTHOR-CREATED ROUTERS

If the system router does not meet your needs, you can create your own router. Remember, when considering the creation of a router, that:

- An author-created router must be programmed; this takes time.
- ECS is charged for the entire router at all times, even when a lesson is being executed.

Thus, it is wise to use the system router when possible.

SUMMARY

To use the system router, you must:

- Ask your account director to create a group and an instructor file
- Specify change and inspect codes for the group
- Specify change and use codes for the instructor file
- Put your lessons in the catalog
- Specify lessons to form modules

A trade-off between catalog and module space must be considered. You are charged only fifty words of ECS for the system router. To create your own router, you must program the router. You are charged ECS for the entire router.

If you would like more information on routers, request the TUTOR feature *routers* in lesson *aids*, or, in the future, when in an actual programming situation, ask a consultant.

