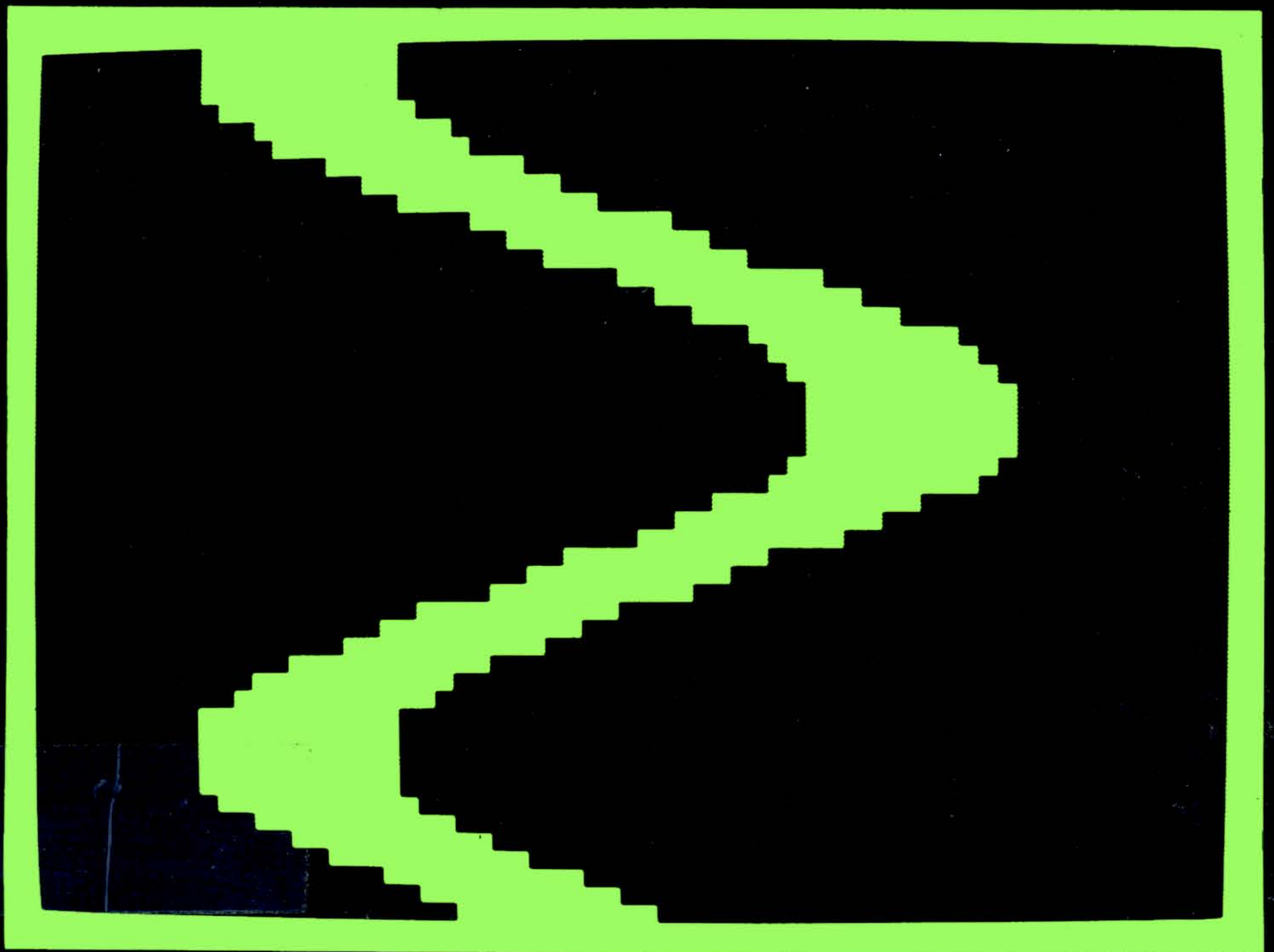
THE TRS-80 GRAPHICS BOOK

Dennis F. Tanner



THE TRS-80 GRAPHICS BOOK

THE TRS-80 GRAPHICS BOOK

Dennis F. Tanner

VAN NOSTRAND REINHOLD COMPANY

THE TRS-80 GRAPHICS BOOK

Copyright © 1984 by Dennis F. Tanner

Library of Congress Catalog Card Number: 84-7524
ISBN: 0-442-28300-8
ISBN: 0-442-28299-0 pbk.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the author.

Manufactured in the United States of America

Published by Van Nostrand Reinhold Company Inc.
135 West 50th Street
New York, New York 10020

Van Nostrand Reinhold Company Limited
Molly Millars Lane
Wokingham, Berkshire RG11 2PY, England

Van Nostrand Reinhold
480 Latrobe Street
Melbourne, Victoria 3000, Australia

Macmillan of Canada
Division of Gage Publishing Limited
164 Commander Boulevard
Agincourt, Ontario M1S 3C7..Canada

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

Library of Congress Cataloging in Publication Data

Tanner, Dennis F.

The TRS-80 graphics book.

Includes index.
1. Computer graphics. 2. TRS-80 (Computer)--Programming. I. Title. II. Title: T.R.S.-80 graphics book.
T385.T36 1984 001.64'43 84-7524
ISBN 0-442-28300-8
ISBN 0-442-28299-0 (pbk.)

Preface

In business, educational and personal software, the addition of graphics to a computer program can make a significant difference in the attractiveness and user-friendliness of a program. For this reason, programming graphics can be quite rewarding.

But it can also be difficult. Determining which pixel to set and which character to print to create a graphic may take hours. Therein lies the graphics programmer's dilemna: spending too much time, or foregoing nicer graphic displays.

The TRS-80 Graphics Book is directed at the graphics programmer. It gives a thorough introduction to graphics programming in BASIC with many carefully explained sample programs. It covers several different techniques, each with its advantages and disadvantages. Then the book introduces the programmer to assembly language graphics programming, showing the programmer how to use machine language programs even without special Editor/Assembler software.

To help the graphics programmer resolve his dilemna, The TRS-80 Graphics Book contains tools. Some of its sample programs include a keyboard input routine, screen editors, graphics file storage programs, animation routines, and more.

For the novice or for the advanced graphics programmer, The TRS-80 Graphics Book provides clear information to assist in making TRS-80 programs come alive with computer graphics.

Acknowledgements

I want to acknowledge the following people for their encouragement during the preparation of this book:

My family (Mom, Bob, Jean, Allan, and Dwight) for their continued interest in the project.

The members of Brentwood Bible Church for their encouragement in this work and (more importantly) in His work.

Most of all, my sweet wife Nicole who spent much of our first year of marriage listening to the "tap, tap, tap" of the keyboard and the whirring of the disk drives.

Dennis Tanner

Contents

Preface/v

Acknowledgments/vii

Chapter 1. Introduction to TRS-80 Graphics Techniques: SET/RESET/POINT GRAPHICS/1

Chapter 2. Introduction to TRS-80 Graphics Techniques: PRINT GRAPHICS/23

Chapter 3. Introduction to TRS-80 Graphics Techniques: POKE GRAPHICS AND THE VIDEO MEMORY/47

Chapter 4. First Applications: Business and Education/69

Chapter 5. Printer Graphics/91

Chapter 6. Using the Variable Pointer to Create Graphics/112

Chapter 7. Using DEBUG Under TRSDOS to Pack String Graphics/134

Chapter 8. Introduction to Assembly Language Graphics Programming/158

Chapter 9. More Graphics in Assembly Language/178

Chapter 10. Assembly Language File Handling in Graphics Programming/202

Appendix 1. Video Display Worksheet/227

Appendix 2. Text Characters and Codes/228

Appendix 3. Graphic Characters/232

Index/233

THE TRS-80 GRAPHICS BOOK

Chapter 1.
Introduction to
TRS-80 Graphics Techniques:
SET/RESET/POINT GRAPHICS

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write and execute a functional example of SET/RESET/POINT graphics, using the proper BASIC syntax and appropriate numbers for X and Y coordinates.
B. Write the upper and lower limits of the horizontal and vertical coordinates of the SET/RESET/POINT statement.
C. Using the SET and RESET instructions, write a program that draws a horizontal line on the screen, then erases it.
D. Using the SET instruction, write a program that draws a vertical line on the screen.
E. Using the SET instruction, write a program that draws a diagonal line on the screen.
F. Using the SET instruction, write a program that draws a circle on the screen.
G. Using the POINT instruction, write a program that determines and displays whether a certain set of graphic points on the screen is SET or RESET.

INTRODUCTION

The simplest and most direct way to program graphics on the TRS-80 Model I and Model III is with the SET/ RESET/POINT instructions. These instructions work on an X,Y coordinate system similar to that used for drawing graphs on graph paper. Do you remember the graphs back in Algebra I that looked like this?

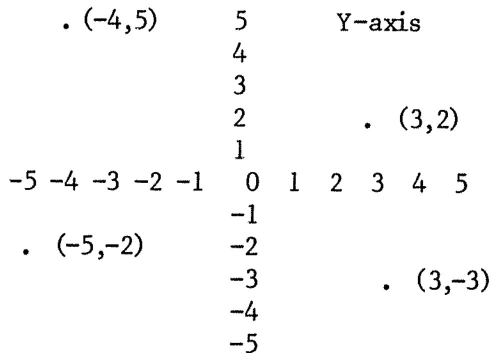


Figure 1-1

Figure 1-1 shows two axes (that's the plural of axis, not of axe). You remember that an axis is a line, right? The horizontal axis is called

the "X-axis" and the vertical axis is called the "Y-axis." This pair of perpendicular axes forms a graph that is appropriately called an "X,Y coordinate system" or a "Cartesian coordinate system." Point (0,0), the center point of the graph, is called the "origin."

Each point on the graph is assigned a pair of coordinates (numbers) that label it as a unique point. The X coordinate comes first, then the Y coordinate. To find the point (-4,5) on the graph, you start at the origin, travel -4 (that is, 4 units to the left) on the X-axis. Then you travel 5 (that is 5 units up) on the Y-axis. Check the positions of the other labeled points in Figure 1-1 to be sure you understand the X,Y coordinate system of labeling points.

So how does this relate to SET/RESET graphics? These graphics also work on an X,Y coordinate system. Imagine your TRS-80 video display as a set of points that begin at the upper left corner of the screen with the point (0,0) and goes to the lower right with the point (127,47). What you are imagining is the exact layout of the TRS-80 screen.

The Video Display Worksheet in Appendix 1 shows the general layout of a TRS-80 Model I or Model III screen for graphics. The numbers closest to the grid show the X and Y coordinates. You may notice three differences between this layout and the Cartesian coordinate system shown above:

1. Only zero and the positive numbers are shown on this layout.
2. The Origin (that's point (0,0), remember?) is in the upper left corner instead of the center of the graph.
3. The X-axis still has the larger numbers to the right, but the Y-axis on this layout is the opposite of the Cartesian coordinate system. It has the small numbers at the top of the screen and the larger numbers at the bottom.

Take a moment to look at the TRS-80 screen. Think about the upper left corner as point (0,0), the upper right corner as point (127,0), the lower left corner as point (0,47), and the lower right as point (127,47).

SETTING POINTS

We will now look at some examples of the SET/RESET/POINT instruction. If you are using a TRS-80 Model I or Model III with disk drives, first type

B A S I C ENTER

to get into Disk BASIC. If your TRS-80 does not have disk drives, you may just begin entering the program.

Let's first look at the SET instruction. To SET (light up) the point in the upper left corner of the screen, type in these instructions:

```
C L S ENTER
S E T ( 0 , 0 ) ENTER
```

Since you are typing instructions without line numbers, you are in the "Immediate Mode" and your instructions are executed immediately. The first instruction clears the screen, and the second turns on the point.

The program in Listing 1-A SETs all four corners of the screen. When you type it in, be sure to press ENTER at the end of each line.

```
10 CLS
20 SET (0,0)
30 SET (127,0)
40 SET (0,47)
50 SET (127,47)
60 GOTO60
```

Listing 1-A

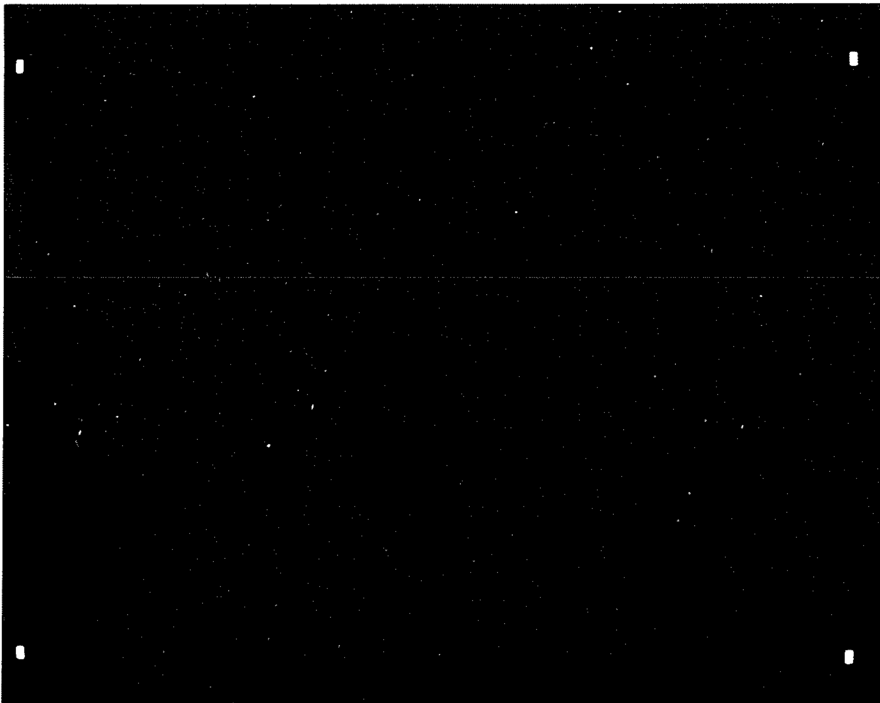


Photo 1-A

Type RUN and press the ENTER key to run this program. Lines 20 through 50 SET the upper left, upper right, lower left, and lower right corners respectively. Line 60 creates an endless loop that supresses the "Ready" prompt. (Try the program without line 60, and you'll see why this is necessary.) To escape this endless loop, press the BREAK key.

To summarize, the general format of the SET instruction is

SET (x,y)

where x is a number between 0 and 127 inclusive, and y is a number between 0 and 47 inclusive.

SETTING LINES

To SET a line, just put the SET instruction in a FOR-NEXT loop, as in Listing 1-B.

```
10 CLS
20 FOR J=0 TO 127
30 SET (J,30)
40 NEXT J
```

Listing 1-B

(When entering this program, remember to press ENTER and the end of each line, and to type RUN and press ENTER to see the program run.)

This program sets every point in a line from (0,30) to (127,30). The result is a horizontal line at vertical position 30, about two thirds of the way down the screen. Why isn't another endless loop such as "50 GOTO 50" needed in this example?

How about a vertical line? Can you think of which elements in this program would have to change? First, the loop in line 20 would go to only 47 instead of 127. Second, the SET statement in line 30 would have the Y coordinate as the looping variable instead of the X coordinate. Thus, the program listing would look like this:

```
10 CLS
20 FOR J=0 TO 47
30 SET (30,J)
40 NEXT J
50 GOTO 50
```

Listing 1-C

This time we have inserted an endless loop at line 50 to inhibit the "Ready" prompt. When you press the BREAK key, you'll see why!

We've gone from SETting one point to SETting a line -- from zero

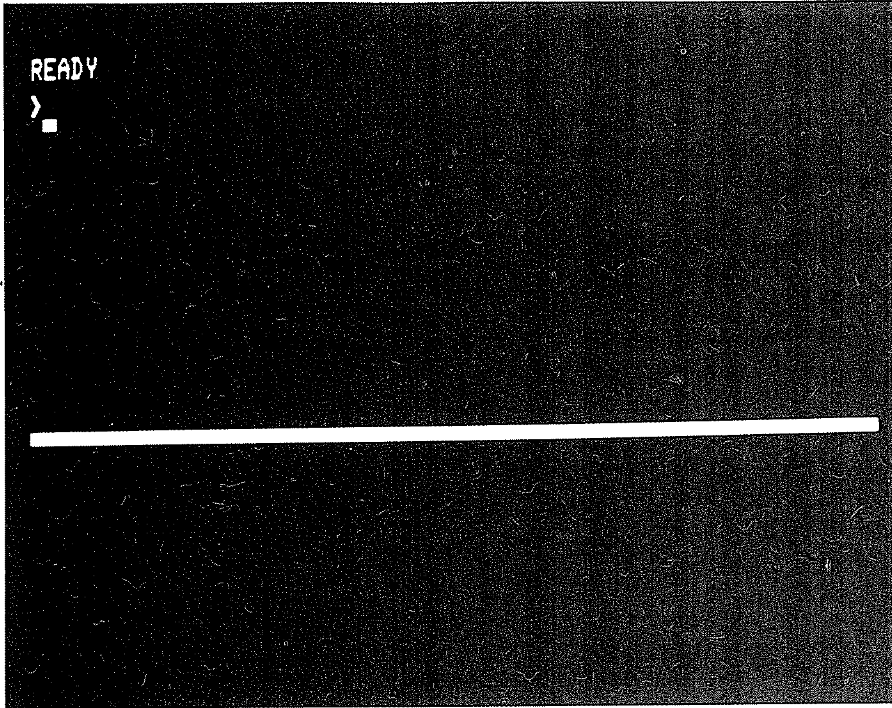


Photo 1-B

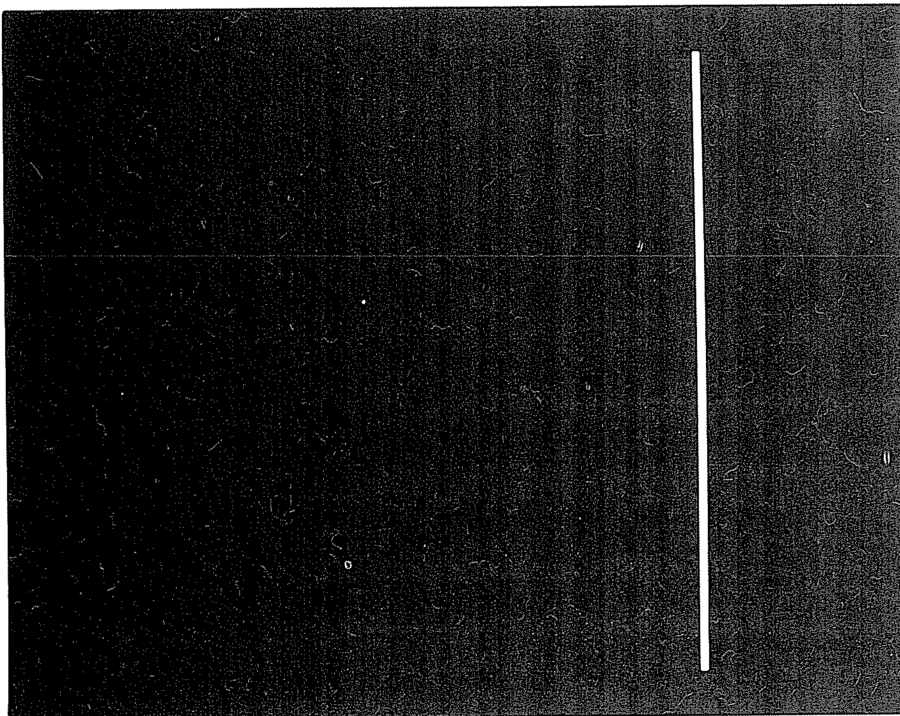


Photo 1-C

dimensions to one dimension? How about proceeding to two dimensions?

To fill in an **area** with the SET instruction, we repeat a vertical line 128 times from left to right until the entire screen is full. Listing 1-D SETs a vertical line each time the loop in lines 30 to 50 is executed. The "J" loop causes lines 30 to 50 to be repeated 128 times, once for each of the horizontal positions. Can you revise the program to SET horizontal lines instead of vertical lines?

```
10 CLS
20 FOR J=0 TO 127
30     FOR K=0 TO 47
40     SET (J,K)
50     NEXT K
60 NEXT J
70 GOTO 70
```

Listing 1-D

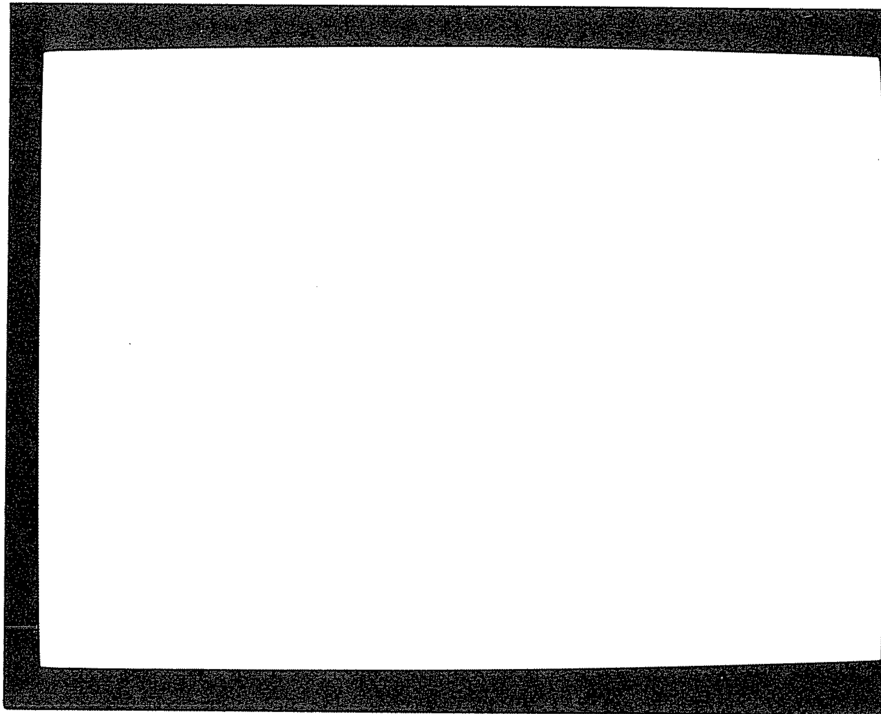


Photo 1-D

Remember that you can press the BREAK key to escape from the loop in line 70.

RESETTING POINTS AND LINES

SETting points light them up or turns them on. In a similar way, RESETting points turns them off.

If you can turn a point on and turn it off, you can make if blink, right? This is demonstrated in the program in listing 1-E. You may notice that this program is identical to that in Listing 1-C except for one line. Line 35 contains the RESET instruction which turns the point off. Before you enter and run this program, can you figure out what it will do?

```
10 CLS
20 FOR J=0 TO 47
30 SET (30,J)
35 RESET (30,J)
40 NEXT J
50 GOTO 50
```

Listing 1-E

Since the program in Listing 1-C draws a vertical line from the top of the screen to the bottom, the program in Listing 1-E turns on the same points with the SET instruction and then turns them off with the RESET in line 35. If this program executes too quickly for you to see it very well, you can add this line:

```
32 FOR I=1 TO 20: NEXT I
```

and change line 50 to this:

```
50 GOTO 20
```

Line 32 adds a delay loop to slow the program down, and line 50 makes lines 20 through 40 repeat.

Would you like to add use some randomness with these SET and RESET instructions? BASIC has an instruction spelled **RND** that selects random numbers. We can use it to SET and RESET points randomly on the screen. The program in Listing 1-F chooses a random X and Y value (in lines 20 and 30) and SETs the point in line 40. Line 50 causes a delay, and line 60 RESETs the point. Line 70 returns the program to line 20 to do it again.

```
10 CLS
20 X = RND(127)
30 Y = RND(47)
40 SET (X,Y)
50 FOR I=1 TO 50: NEXT I
```

```
60 RESET (X,Y)
70 GOTO 20
```

Listing 1-F

The program can be made to RESET about half the points and leave the other half SET by changing line 60 to read as follows:

```
60 IF RND(2) = 1 THEN RESET (X,Y)
```

If RND(2) returns 1, the point will be RESET. If it returns 2, the program will continue to line 70 without RESETting the point.

To summarize, the general format of the RESET instruction is

RESET (x,y)

where **x** is a number between 0 and 127 inclusive, and y is a number between 0 and 47 inclusive.

SETTING DIAGONAL LINES

So far we have created horizontal and vertical lines by changing either the X or the Y value in the SET instruction. Diagonal lines can be created on your TRS-80 microcomputer by changing the X and Y values simultaneously in the SET (X,Y) instruction.

```
10 CLS
20 FOR J=0 TO 47
30    X=J
40    Y=J
50    SET (X,Y)
60 NEXT J
70 GOTO70
```

Listing 1-G

The program in Listing 1-G SETs each point along a line from (0,0) to (47,47). The X and the Y values are incremented simultaneously.

```
10 ON ERROR GOTO 170
20 CLS
30 INPUT "START POINT (X,Y)"; SX, SY
40 INPUT "UP OR DOWN"; UD$
50 IF UD$ <> "UP" AND UD$ <> "DOWN" THEN 40
60 IF UD$ = "UP" THEN IC = -1 ELSE IC = 1
70 INPUT "HOW MANY POINTS"; NP
80 INPUT "INCREMENT Y FOR EACH X"; IY
90 IC = IC * IY
100 FOR J = 0 TO NP - 1
110    X = SX + J
```

```
120    Y = SY + J * IC
130    SET (X,Y)
140 NEXTJ
150 I$ = INKEY$
160 I$ = INKEY$: IF I$ = "" THEN 160 ELSE 20
170 PRINT @ 960, "ERROR";
180 RESUME NEXT
```

Listing 1-H

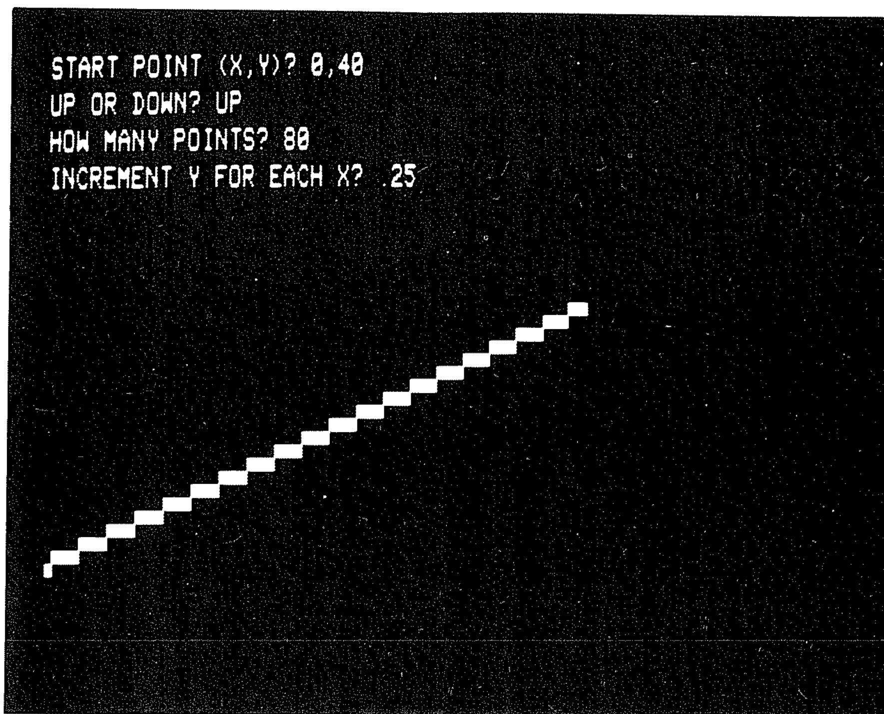


Photo 1-H

The program in Listing 1-H is a more general program that allows the user to INPUT the following information at the keyboard:

Information	Variable
Starting point (X)	SX
Starting point (Y)	SY
Up/down variable	UD$
Number of points	NP
Increment of Y for each X	IY

Other variables used are as follows:

Information	Variable
Increment (1 or -1)	IC
Looping variable	J
X value	X
Y value	Y
Inkey$ variable	I$

This program lets you see the effect of various angles on the lines in the TRS-80 low-resolution graphics.

For those who are interested, here is a line-by-line description of the functions of this program:

Line	Description
10	Sets up an error-trapping routine. If a point is out of range, the program will jump to line 170.
20	Clears the screen.
30	Accepts the starting point from the keyboard (SX,SY).
40	Accepts UP or DOWN from the keyboard (UD$).
50	If UD$ is not "UP" or "DOWN," the program jumps back to line 40 for more keyboard input.
60	If the user typed "UP," the variable IC (increment) is set at -1. Otherwise it is set at 1. This value is used in lines 90 and 120 to determine the Y coordinate.
70	Accepts the number of points to be plotted from the keyboard (NP).
80	Accepts the value to use in computing the increment of Y for each X step at the keyboard (IY). This value is used in line 90. If the value is 1, the Y will be incremented 1 for every X across. If the value is 2, Y will be incremented 2 for every X across, resulting in a sharper slope. If the value is .5, the slope is more gradual. This value, along with IC, determines the slope in line 90.
90	The new IC value is computed using the previous IC value and IY.
100	Begins a FOR-NEXT loop using the variable J. The loop is executed NP times, starting with 0 and going to NP-1.
110	The X value is computed as the start value plus the loop value.
120	The Y value is computed as the start value plus the loop value times the increment value.
130	The point is SET.
140	This line ends the FOR-NEXT loop.
150	INKEY$ catches keystrokes. This INKEY$ catches any key presses that might have been made during the drawing of the line.
160	This second INKEY$ catches intentional keystrokes. If no key was pressed (and therefore I$="") the program loops on line 160. If a key was pressed, the program goes back to line 20.
170	This begins the error-trapping routine. The work "ERROR" is printed in the bottom left corner. The semicolon after the second quotation mark supresses the linefeed and prevents the screen from

scrolling.

180 After "ERROR" is printed, the program RESUMEs at the line following the error.

By playing with this program, you can see the effect of different slopes is on the low-resolution line.

The following changes can be made to the program to see a random display of lines on the screen using the SET instruction. Use these new lines to replace the corresponding lines in Listing 1-H:

```
10 ONERROR GOTO 190
30 SX = RND (50): SY = RND (47)
40 IF RND (2) = 1 THEN UD$ = "UP"
   ELSE UD$ = "DOWN"
(DELETE LINE 50)
70 NP = RND (127 - SY)
80 IY = RND (0) * 2
95 FOR K = 1 TO 2
130 IF K = 1 THEN SET (X,Y) ELSE RESET (X,Y)
145 NEXT K
150 GOTO 20
190 J = NP - 1: RESUME NEXT
```

A subroutine similar to this can be used within a program to draw lines of varying slope and length.

Listing 1-I contains a another program that draws lines. This one uses both endpoints instead of one endpoint and the slope. Can you figure out how this program SETs lines given the endpoints?

```
10 CLS
20 INPUT "START (X,Y)"; SX,SY
30 INPUT "   END (X,Y)"; EX,EY
40 LE = SQR ((EX-SX) 2 + (EY-SY) 2)
50 IF LE = 0 THEN PRINT "LENGTH IS ZERO":
   GOTO 110
60 FOR J = 0 TO LE
70     X = SX + (EX-SX) * J/LE
80     Y = SY + (EY-SY) * J/LE
90     SET (X,Y)
100 NEXT J
110 I$ = INKEY$
120 I$ = INKEY$: IF I$ = "" THEN 120
130 GOTO 10
```

Listing 1-I

Note that this program first computes the length of the line, then SETs a point for each number up to that length.

This program could also be made in to a subroutine that would SET lines for given endpoints within a program.

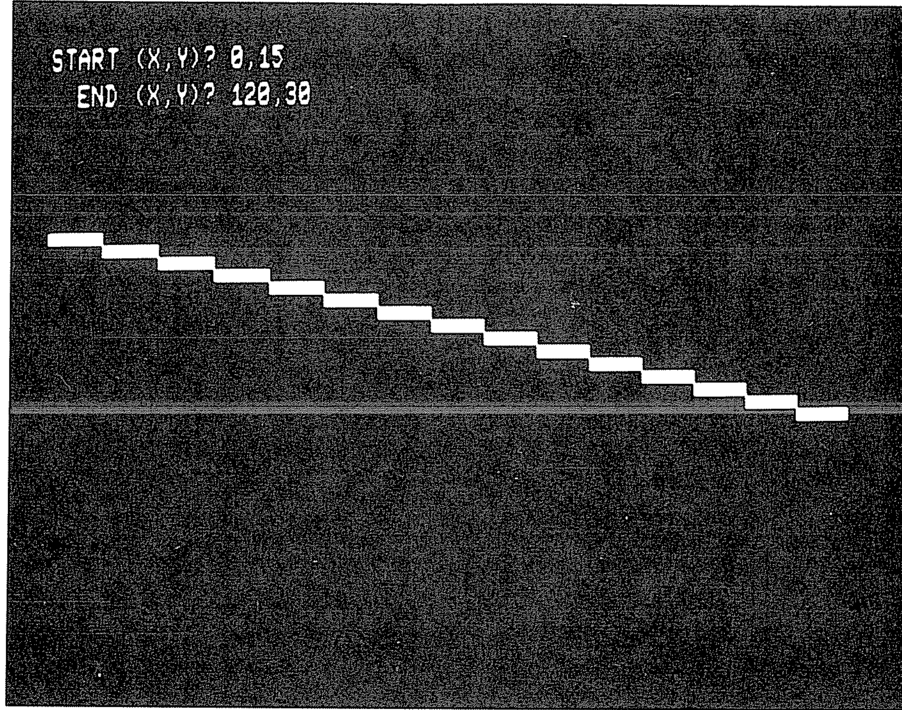


Photo 1-I

SETTING CIRCLES

The TRS-80 Model I and Model III have built-in geometric functions which facilitate the creation of graphic circles. If you consider the triangle in Figure 1-2, you can see that the **cosine** function represents the ratio of the adjacent side (side AC) to the hypotenuse (side AB). The **sine** function is the ratio of the opposite side (side BC) to the hypotenuse. Using these ratios, we can set up a formula for determining the location of any point on the circle, given its angle.

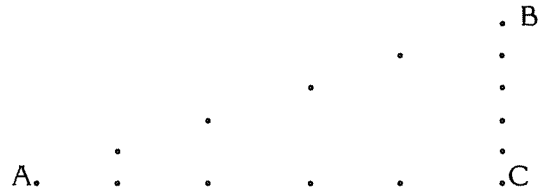


Figure 1-2

Cosine (Angle BAC) = Adjacent / Hypotenuse
= Side AC / Side AB

Sine (Angle BAC) = Opposite / Hypotenuse =
= Side BC / Side AB

In any triangle on the circle with radius AB, the horizontal (X) coordinate is computed using the cosine function, and the vertical (Y) is computed using the sine function.

That's enough theory for a while. Listing 1-J shows a program that draws a circle with the SET instruction, using the geometric functions as described.

```
10 CLS
20 FOR AN = 0 TO 6.3 STEP .02
30     X = 64 + COS (AN) * 40
40     Y = 24 + SIN (AN) * 18
50     SET (X,Y)
60 NEXT AN
70 GOTO 70
```

Listing 1-J

Here are the variables used in the program:

Variable	Meaning
AN	The angle in radians
X	The X coordinate
Y	The Y coordinate

And here is a line-by-line description of the program:

Line	Description
10	Clear the screen
20	Begins a loop that creates the circle. The loop is up to 6.3 (approximately 2 X Pi), which is the number of radians in a circle. The value of AN is the current angle of the circle. The STEP value is .02 to draw the circle with a large number of increments. (If no step value were given, there would be only seven points in the circle!)
30	X is computed. 64 is about half of 127 (the largest X coordinate) so the circle will be centered horizontally. The cosine of the angle is multiplied by 40 so the radius will be approximately 40 graphic units.
40	Y is computed. 24 is about half of 47 (the largest Y coordinate) so the circle will be centered vertically. The sine of the angle is multiplied by 18. This value is smaller than the 40 used in line 30 to compensate for the graphic point being taller than it is wide.

50 The point is SET.
60 The loop is closed.
70 This keeps the "Ready" prompt from appearing until the BREAK key is pressed.

The preceding description contains ideas that will be used in the following programs, so it would be good to digest it before proceeding.

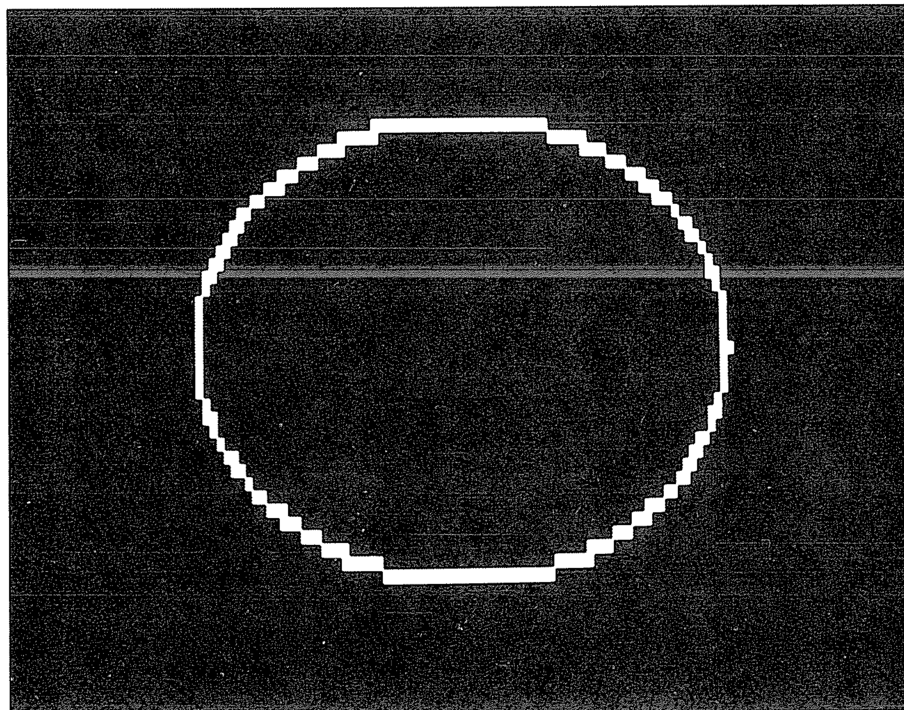


Photo 1-J

Listing 1-K shows a general circle- and ellipse-drawing program. It allows the user to choose the center (defined by BEGINNING X and BEGINNING Y) and the radius. A complete explanation is left to the reader.

```
10 ON ERROR GOTO 160
20 CLS
30 INPUT "RADIUS"; RA
40 INPUT "BEGINNING X"; X1
50 INPUT "BEGINNING Y"; Y1
60 INPUT "STEP"; ST
70 INPUT "X:Y RATIO"; RT
80 FOR AN = 0 TO 6.3 STEP ST
90      X = X1 + COS (AN) * RA
```

```
100     Y = Y1 + SIN (AN) * RA * RT
110     SET (X,Y)
120 NEXT AN
130 PRINT "DONE";
140 I$ = INKEY$
150 I$ = INKEY$: IF I$ = "" THEN 150 ELSE 10
160 RESUME NEXT
```

Listing 1-K

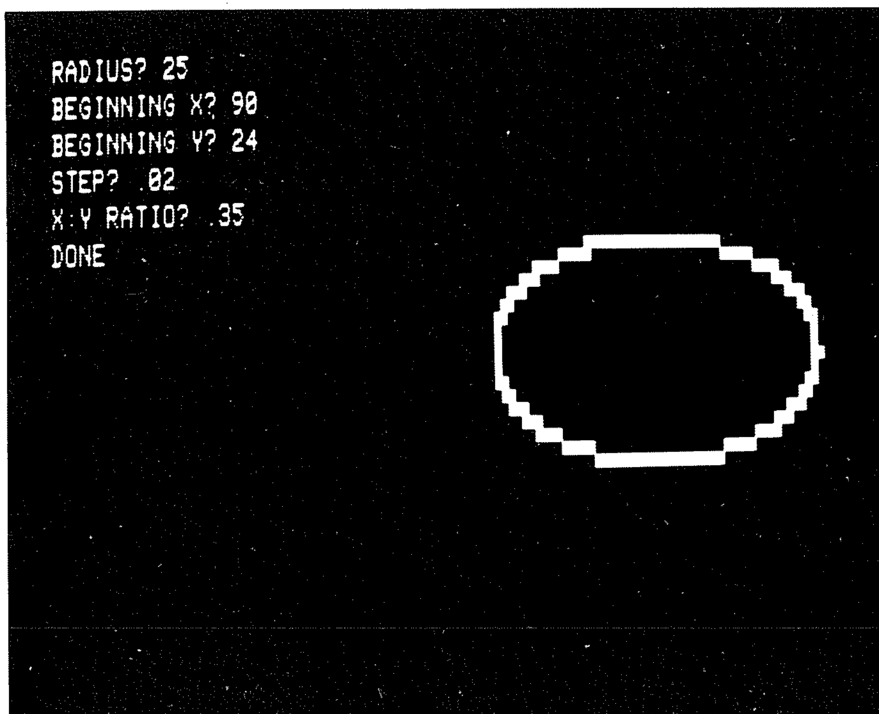


Photo 1-K

For a little fun, the program in Listing 1-L draws a random but pleasing pattern of concentric circles. The program occasionally clears the screen so it won't become too cluttered.

```
10 ON ERROR GOTO 150
20 CLS
30 RA = RND (60)
40 X1 = 64
50 Y1 = 24
60 ST = .6 / RA
70 RT = .5
```

```
80 FOR AN = 0 TO 6.3 STEP ST
90     X = X1 + COS (AN) * RA
100    Y = Y1 + SIN (AN) * RA * RT
110    SET (X,Y)
120 NEXT AN
130 IF RND (30) = 1 THEN CLS
140 GOTO 30
150 RESUME NEXT
```

Listing 1-L

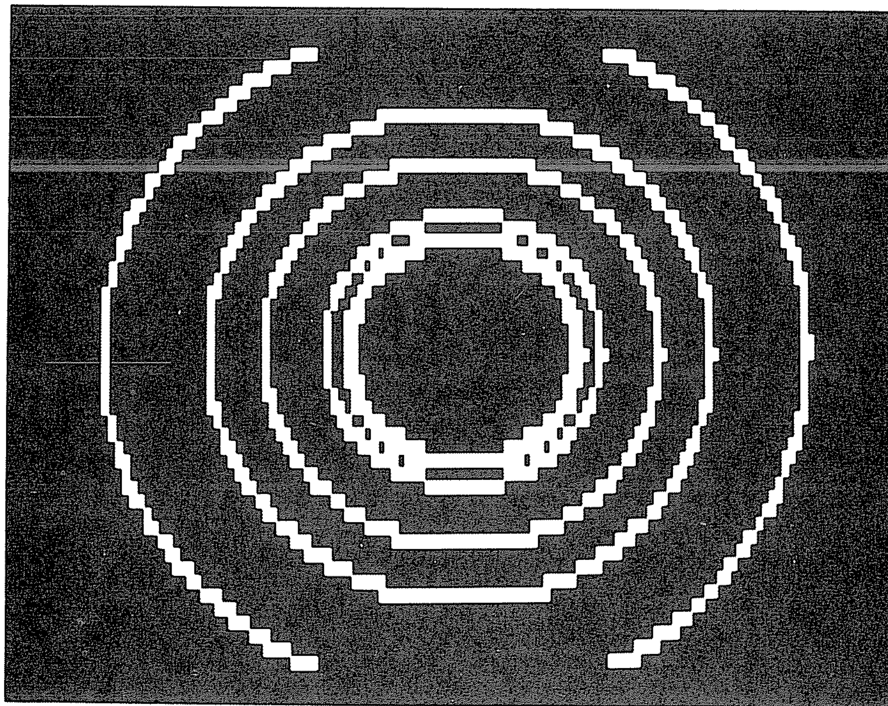


Photo 1-L

THE POINT INSTRUCTION

The POINT instruction determines whether a graphics point is SET or not. It can be used in BASIC programs to tell what graphics are on the screen. The POINT instruction returns the value 0 if the graphics point is not SET and -1 if the point is SET.

Listing 1-M shows a program that demonstrates the POINT instruction.

```
10 CLS
20   FOR X = 1 TO 4
30 SET (X, 30)
40 NEXT X
50 FOR X = 1 TO 8
60   PRINT "POINT (" X ",30) =" POINT (X, 30)
70 NEXT X
80 GOTO 80
```

Listing 1-M

Lines 20 through 40 SET a short line of points. Lines 50 to 70 use the POINT instruction to determine and display whether each point is SET. For the first point, the display will read,

POINT (1 ,30) = -1

This means the first point is SET. For the last point, the display will read,

POINT (8 ,30) = 0

This means the last point is **not** SET.

Another way to use the value returned by the POINT instruction is as a true or false value. The program in Listing 1-N shows how to do this.

```
10 CLS
20 FOR X = 1 TO 4
30   SET (X, 30)
40 NEXT X
50 FOR X = 1 TO 8
60   PRINT "POINT (" X ",30) IS ";:
     IF POINT (X, 30) THEN PRINT "SET"
     ELSE PRINT "NOT SET"
70 NEXT X
80 GOTO 80
```

Listing 1-N

Again, lines 20 to 40 SET the graphics points in the line. The loop in lines 50 to 70 determine whether each of the first eight points is SET. Since the first four points are SET, their POINT value is true and will be reported as "SET." The POINT value returned for the last four points is false, so they will be reported as "NOT SET." when the program runs.

How can this POINT instruction be used in a graphics program? The program in Listing 1-O shows an example.

```
5 ON ERROR GOTO 100
10 CLS
20 FOR X = 0 TO 127: SET (X, 0): SET (X, 47):
       NEXT X
30 FOR Y = 0 TO 47: SET (0, Y): SET (127, Y):
       NEXT Y
```

```
40 X = RND (127): Y = RND (47)
50 XA = RND (3) - 2: YA = RND (3) - 2
60 X = X + XA: Y = Y + YA
70 IF POINT (X, Y) THEN 50
80 SET (X, Y)
90 GOTO 60
100 RESUME 40
```

Listing 1-O

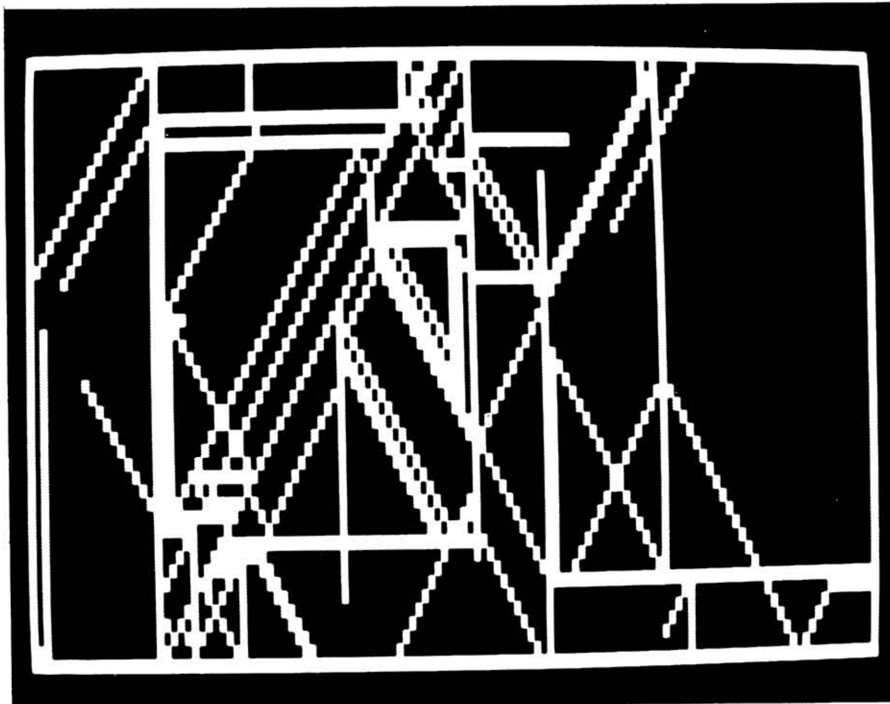


Photo 1-O

Here's a list of variables used in this program:

Variable	Meaning
X	The X coordinate in SET and POINT statements
Y	The Y coordinate in SET and POINT statements
XA	Number by which the X coordinate is incremented
YA	Number by which the Y coordinate is incremented

And here are the routines used in Listing 1-O:

Lines	Description
5	Sets up error-trapping routine (in case a set of (X,Y) values is beyond those that can be used in a SET or POINT instruction)
10	Clears the screen
20	Draws the horizontal border lines
30	Draws the vertical border lines
40	Chooses random start values for X and Y
50	Chooses random value for XA and YA. The values will be either -1, 0, or 1.
60	The current X value is incremented by XA, and and the current Y value is incremented by YA. This causes the line to "move" in a direction determined by the XA and YA values.
70	**Here's the POINT instruction.** If the graphics point that is about to be SET is ALREADY set, a different increment (and therefore a different direction) is chosen. The effect is that each time the line intersects an existing line, there is a good chance it will change direction. This creates an attractive visual effect.
80	The next graphics point is SET.
90	The program loops back for the next point on the line.
100	This is the error-trapping routine. Whenever the X or Y value in the POINT instruction for the next location is beyond the acceptable limits, the program jumps to this line which in turn returns the program to line 40 for another random start point.

Chapter Summary

The screen is divided into graphics points numbered from 0 to 127 going from left to right, and from 0 to 47 going from top to bottom.

The SET instruction is used to "turn on" a graphics point on the screen. Its syntax is **SET (x,y)** where **x** is a number between 0 and 127, and **y** is a number between 0 and 47.

The RESET instruction is used to "turn off" a graphics point on the screen. Its syntax is **RESET (x,y)** where **x** is a number between 0 and 127, and **y** is a number between 0 and 47.

Horizontal and vertical lines can be drawn on the screen using the SET instruction in a FOR-NEXT loop which changes either the **x** or **y** element in the instruction.

Diagonal lines can be drawn on the screen using the SET instruction in a FOR-NEXT loop which changes both the **x** and the **y** element in the instruction.

Circles can be drawn on the screen using the SET instruction in a FOR-NEXT loop and the sine and cosine functions.

The POINT instruction determines whether a certain graphics point is SET or RESET. Its syntax is **POINT (x,y)** where x is a number between 0 and 127 and y is a number between 0 and 47.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Write and execute a functional example of SET/RESET/POINT graphics, using the proper BASIC syntax and appropriate numbers for X and Y coordinates.
B. Write the upper and lower limits of the horizontal and vertical coordinates of the SET/RESET/POINT statement.
C. Using the SET and RESET instructions, write a program that draws a horizontal line on the screen, then erases it.
D. Using the SET instruction, write a program that draws a vertical line on the screen.
E. Using the SET instruction, write a program that draws a diagonal line on the screen.
F. Using the SET instruction, write a program that draws a circle on the screen.
G. Using the POINT instruction, write a program that determines and displays whether a certain set of graphic points on the screen is SET or RESET.

More Programming Practice

1. Using the SET instruction, write a program that draws your name on the screen.

2. Write a program that asks the user for three points, then draws a triangle with those points as corners.

3. Write a program that randomly SETs a line on the screen, then RESETs it in the opposite direction. (Hint: Use a FOR-NEXT-STEP loop.)

4. Write a program that draws a face on the screen using circles for the eyes, nose, mouth, and the outline of the face.

5. Write a program that draws random circles on the screen. An error-trapping routine should be used to prevent errors when a circle would go beyond the screen. When the circles intersect, the point of intersection should be RESET instead of SET.

Chapter Checkup

1. What are the X- and Y-coordinates of the upper right corner of the screen?

2. What does each line of the following program do?

```
10 CLS
20 FOR J = 0 TO 47
30  SET (J,127)
40 NEXT J
50 GOTO 50
```

3. In narrative form, write out the method you would use to draw dark circles on a white screen.

4. How many graphics points are on the TRS-80 screen?

5. When SETTING a horizontal line, which element of the SET instruction varies, the X or the Y?

6. This is the program in Listing 1-E:

```
10 CLS
20 FOR J=0 TO 47
30 SET (30,J)
35 RESET (30,J)
40 NEXT J
50 GOTO 50
```

   How would you change it to move the dot horizontally instead of vertically?

7. What value does the POINT instruction return:
   a) If the graphics point is SET?
   b) If it is RESET?
   c) If the location contains a text character?

8. This program sets a horizontal line:

```
10 CLS
20 FOR X = 0 TO 127
30  SET (X,0)
40 NEXT X
50 GOTO 50
```

   a) How would you change it to fill the entire screen with horizonal lines?
   b) Write and RUN the program. How long does the program take to fill the screen?

9. Write a line-by-line description of how program 1-I SETs graphic lines on the screen.

10. Run the following program:

```
10 CLS
20 FOR X=0 TO 10
30  FOR Y=0 TO 10
40   SET (X,Y)
50  NEXT Y
60 NEXT X
70 PRINT "A"
80 GOTO 80
```

(a) How many pixels (graphic dots) does the letter "A" displace?
(b) How many pixels vertically?
(c) How many pixels horizontally?
(d) Since there are 128 graphics positions across the screen, how many text positions are there across?
(e) Since there are 48 graphics positions down the screen, how many text positions are there down?

Chapter 2.
Introduction to
TRS-80 Graphics Techniques:
PRINT GRAPHICS

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write a program that draws a square using the PRINT instruction with an "*".
B. Write a program that PRINTs a box on the TRS-80 screen using PRINT @ and graphics characters.
C. Write a program that PRINTs a box on the TRS-80 screen by assigning graphics characters to a string variable and PRINTing the variable.
D. Write a program using the STRING$ function that fills one line on the TRS-80 screen with a row of graphics characters.
E. Write a program that draws a heart on the screen by assigning graphics characters to a string variable and PRINTing the variable. Use Character 26 and a STRING$ of Character 8 to go to the next line of the screen.

INTRODUCTION

With SET/RESET/POINT graphics, the programmer can turn on, turn off, and determine the status of any of the pixels on the 128 by 48 grid on the TRS-80 screen. Using PRINT graphics, the programmer can place any of the regular text characters as well as any of the graphics characters anywhere in the 1024 "PRINT AT" screen positions. This chapter introduces PRINT positions, graphics characters, and graphics strings.

PRINTING WITH REGULAR TEXT CHARACTERS

Return for a moment to Appendix 1. This Video Display Worksheet shows the X- and Y-coordinates used on the TRS-80 display with the numbers closest to the grid on the page. The Worksheet also shows the **PRINT AT positions** in larger boxes on the left and right sides of the grid. Note that the grid contains 1024 positions, numbered left to right and top to bottom with 0 at the top left and 1023 at the bottom right. These positions can be used with PRINT instructions to define the location of PRINTing on the TRS-80 screen.

Try this program.

```
10 CLS
20 PRINT "THIS IS A TRS-80 COMPUTER."
```

Listing 2-A

This program clears the screen and prints the message in the upper left corner, position 0. The program in Listing 2-B is similar, but this time the position of printing is defined using the **PRINT @** (read "print at") instruction in line 20.

```
10 CLS
20 PRINT @ 140, "THIS IS A TRS-80 COMPUTER."
```

Listing 2-B

Using regular text characters and the PRINT @ instruction, we can create pictures on the display screen. Listing 2-C shows the program that creates the screen shown in Photo 2-I. The "PRINT @" in lines 20 to 160 cause the graphic to begin at the 20th position in from the left of the screen. Note that the left quotation mark in each line is lined up with the others to make the program easier to type and to read. Some characters in this program were chosen because they look like the part of the person they portray. Others, like the "SHOE" in line 160, spell out the word that corresponds with a part of the picture.

```
10 CLS
20 PRINT @ 20,  "       @@@@"
30 PRINT @ 84,  "    @@@@@@@@@@@@"
40 PRINT @ 148, " @@    0  0     @@"
50 PRINT @ 212, " C      L        3"
60 PRINT @ 276, "  (   ======    )"
70 PRINT @ 340, "    -----------"
80 PRINT @ 404, "    ##==-L==##"
90 PRINT @ 468, "  (            .)"
100 PRINT @ 532," !                !"
110 PRINT @ 596," !  !           !  !"
120 PRINT @ 660," WWW            WWW"
130 PRINT @ 724,"    !   !  !   !"
140 PRINT @ 788,"    (   !  !   )"
150 PRINT @ 852,"    !   !  !   !"
160 PRINT @ 916," SHOEIII IIISHOE"
170 GOTO 170
```

Listing 2-C

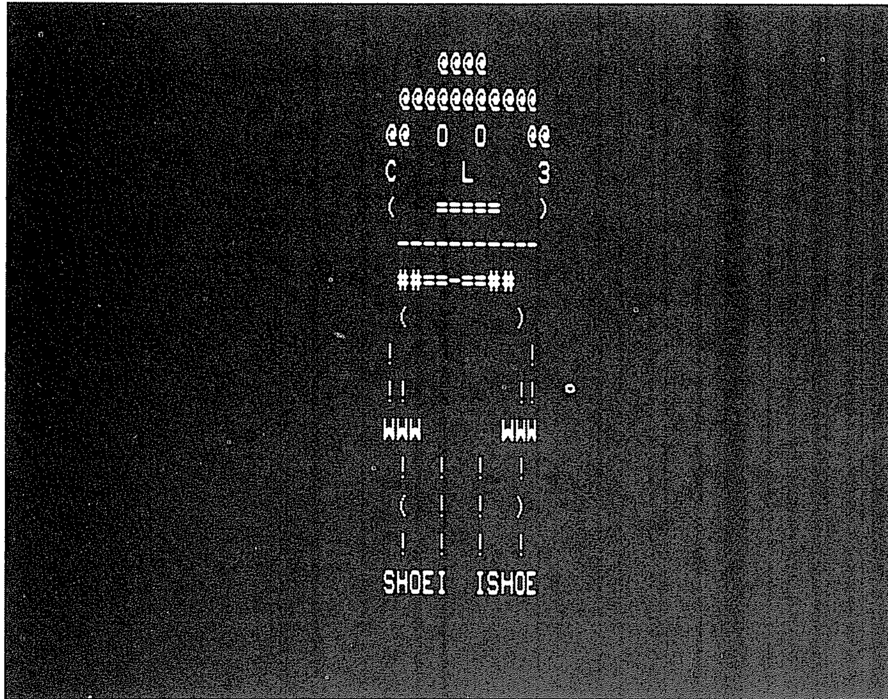


Photo 2-C

Here is another example:

```
10 CLS
20 PRINT "                    LEAVES"
30 PRINT "                 LEAVESLEAVES"
40 PRINT "           LEAVESLEAVESLEAVESLEAVES"
50 PRINT "         LEAVESLEAVES    BIRD    LEAVES"
60 PRINT "        LEAVESLEAVESLEAVESLEAVESLEAVES"
70 PRINT "         LEAVESLEAVESLEAVESLEAVESLEA"
80 PRINT "           LEAVESLEAVESLEAVESLEAVES"
90 PRINT "                 LEAVESLEAVES"
100 PRINT"                    TRUNK"
110 PRINT"                    TRUNK"
120 PRINT"                    TRUNK"
130 PRINT"                    TRUNK"
140 PRINT"                  ROOTSROOTS"
150 GOTO 150
```

Listing 2-D

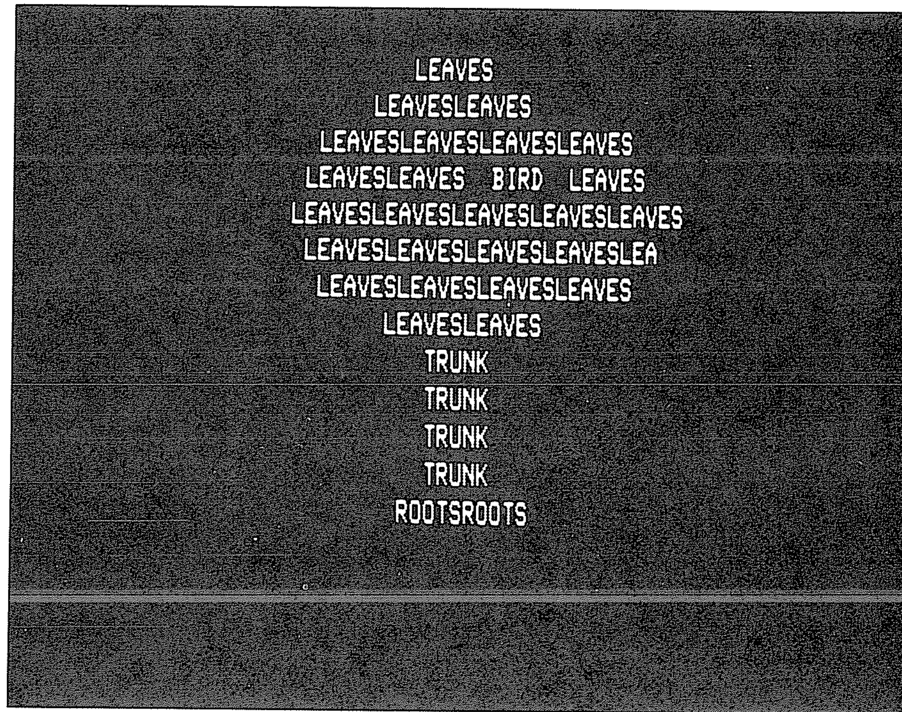


Photo 2-D

This program uses regular PRINT instructions rather than PRINT @. Each PRINT statement begins on a new line, so the spacing inside the quotation marks determines the positions on the display screen.

PRINTING GRAPHIC CHARACTERS

Besides the regular text characters, your TRS-80 Model I or Model III has a set of graphic characters. These characters have numbers that correspond with them, just like the regular text characters do. To PRINT the character that corresponds to a certain number, you may use the CHR$ function. Listing 2-E shows a program that prints a capital "A" in the middle of the screen.

```
10 CLS
20 PRINT @ 544, CHR$(65)
```

Listing 2-E

The number 65 refers to the letter "A" on your TRS-80. Appendix 2 shows all the text characters and their corresponding numbers.

The graphics characters have numbers from 128 to 191. Try the program again, only this time type line 20 as shown in Listing 2-F.

```
10 CLS
20 PRINT @ 544, CHR$(191)
```

Listing 2-F

When you RUN this program, you will see the graphics character corresponding with the number 191 in the center of the screen.

Do you want to see all the graphics characters? The program in Listing 2-G shows all of them.

```
10 CLS
20 FOR J=128 TO 191
30 PRINT J; CHR$(J);
40 IF RIGHT$ (STR$(J),1) = "7" THEN PRINT: PRINT
50 NEXT J
60 GOTO 60
```

Listing 2-G

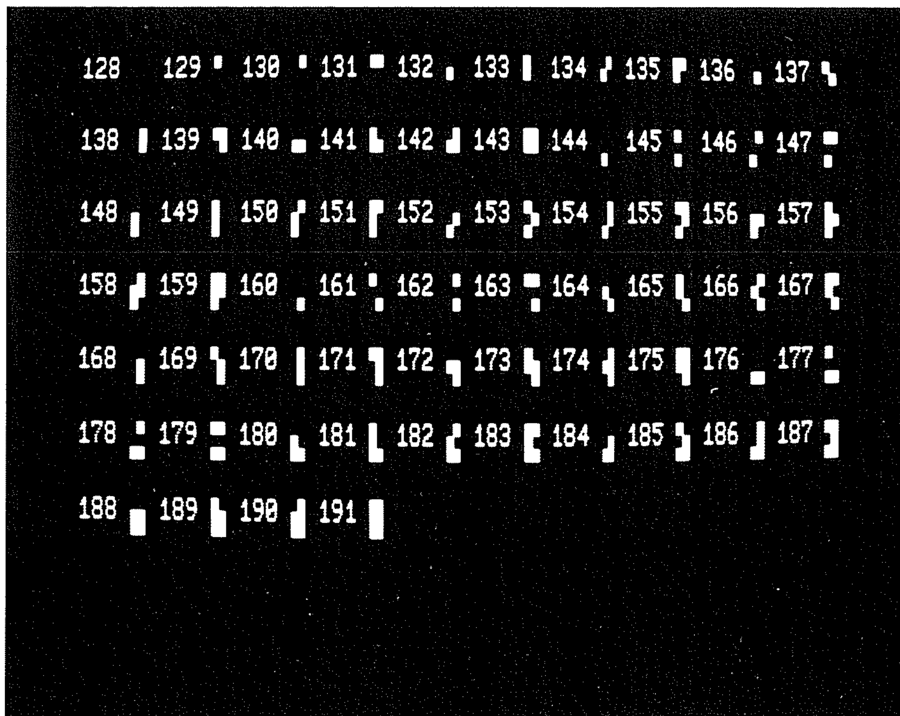


Photo 2-G

Here is a line-by-line description of this program:

Line	Description
10	Clears the screen
20	Begins a loop from 128 to 191, the numbers that correspond to the graphics characters
30	Prints the current loop number, and the character that corresponds to it. Semicolons are used to prevent printing from going to the next line, thus packing more characters on each line of the screen.
40	Checks to be see of the current loop number has "7" as its right digit. If it does, the program executes two PRINTs, one to move to the beginning of the next line, and the other to add a line before PRINTing more characters. Thus, after each ten characters, a blank line is printed. The number "7" was chosen because 137 is the tenth number when counting from 128.
50	Ends the FOR-NEXT loop
60	Creates an endless loop to freeze what is PRINTed on the screen. You can press BREAK to escape this loop.

These characters are also shown in Appendix 3.

Since we can PRINT at any position on the screen with PRINT @, and since we can PRINT any graphics character, we can begin to draw some pictures on the screen. Listing 2-H shows a program that prints graphics characters at certain points on the screen, to form a box.

```
10 CLS
20 PRINT @ 479, CHR$(170);
30 PRINT @ 480, CHR$(131);
40 PRINT @ 481, CHR$(131);
50 PRINT @ 482, CHR$(149);
60 PRINT @ 543, CHR$(130);
70 PRINT @ 544, CHR$(131);
80 PRINT @ 545, CHR$(131);
90 PRINT @ 546, CHR$(129);
100 GOTO 100
```

Listing 2-H

Each line of this program PRINTs one graphics character at one screen position. You can see which graphics character corresponds to each character number used in this program by checking Appendix 3.

Did you notice how fast that box came on the screen when you ran the program in Listing 2-H? As a rule, **PRINT graphics execute more quickly than SET/RESET/ POINT graphics.**

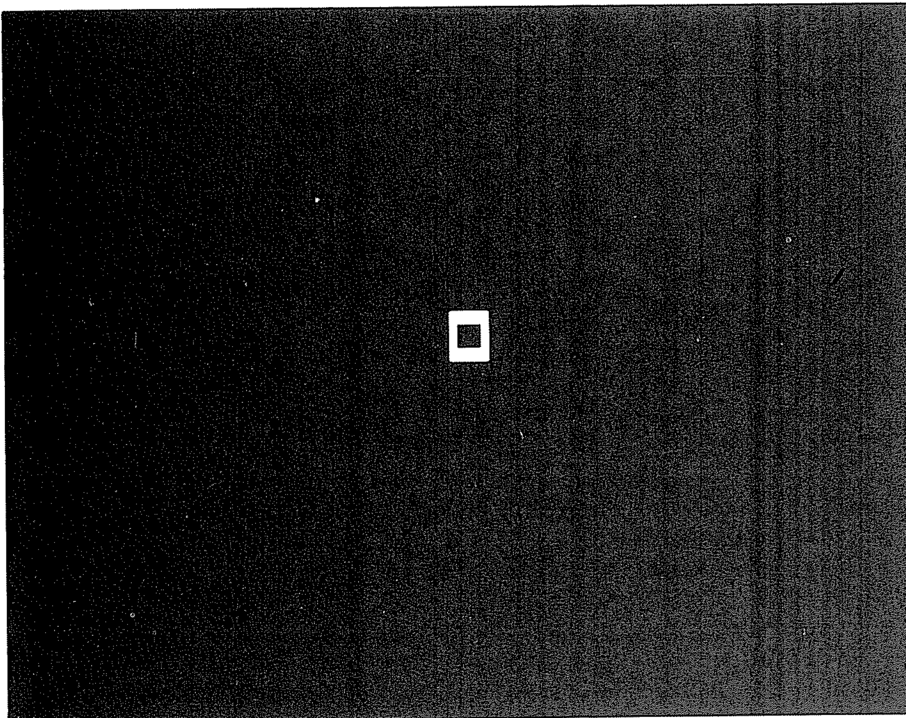


Photo 2-H

Large, complex pictures can be drawn by PRINTing graphics characters on the screen. The program in Listing 2-I shows and example of a complex picture. A line-by-line description of the program follows the listing.

```
10 CLS
20 READ A,B
30 IF A= -1 THEN 1000
40 PRINT @ A, CHR$(B);
50 GOTO 20
60 DATA  87,136,  88,180, 152,170, 212,176
70 DATA  213,176, 214,176, 215,176, 216,178
80 DATA  217,179, 218,189, 219,176, 220,176
90 DATA  221,176, 222,176, 223,176, 224,176
100 DATA 225,176, 226,176, 227,176, 228,176
110 DATA 229,176, 230,176, 231,176, 232,176
120 DATA 233,144, 274,160, 175,158, 176,129
130 DATA 281,156, 282,140, 283,156, 284,148
140 DATA 295,160, 296,158, 297,173, 298,144
150 DATA 337,184, 338,135, 345,181, 346,176
160 DATA 347,159, 348,129, 358,184, 359,135
170 DATA 362,139, 363,180, 399,160, 400,190
```

```
180 DATA 401,177, 402,176, 403,176, 404,176
190 DATA 405,176, 406,176, 407,176, 408,176
200 DATA 409,176, 410,176, 411,176, 412,176
210 DATA 413,176, 414,176, 415,176, 416,176
220 DATA 417,176, 418,176, 419,176, 420,176
230 DATA 421,190, 422,177, 423,176, 424,176
240 DATA 425,176, 426,176, 427,178, 428,189
250 DATA 429,170, 463,170, 464,149, 485,191
260 DATA 492,170, 493,149, 527,170, 528,149
270 DATA 536,156, 537,140, 538,172, 549,191
280 DATA 552,168, 553,140, 554,172, 556,170
290 DATA 557,149, 591,170, 592,149, 600,131
300 DATA 601,131, 602,131, 613, 191 616,170
310 DATA 618,170, 620,170, 621,149, 655,138
320 DATA 656,141, 657,140, 658,140, 659,140
330 DATA 660,140, 661,140, 662,140, 663,140
340 DATA 664,140, 665,140, 666,140, 667,140
350 DATA 668,140, 669,140, 670,140, 671,140
360 DATA 672,140, 673,140, 674,140, 675,140
370 DATA 676,140, 677,140, 678,140, 679,140
380 DATA 680,142, 681,140, 682,142, 683,140
390 DATA 684,142, 685,133, 720,148, 784,157
400 DATA 785,140, 786,172, 788,168, 789,140
410 DATA 790,140, 797,148, 792,168, 795,148
420 DATA 796,168, 797,140, 798,140, 799,132
430 DATA 801,156, 802,140, 803,140, 804,148
440 DATA 848,149, 850,170, 852,170, 853,176
450 DATA 854,176, 859,149, 856,170, 857,176
460 DATA 858,176, 859,149, 860,162, 861,179
470 DATA 862,179, 863,149, 865,183, 866,179
480 DATA 867,179, 868,145
999 DATA -1, -1
1000 GOTO 1000
```

Listing 2-I

Line	Description
10	Clears the screen
20	Reads the first number in the DATA statements (87) into the variable A and the second number (136) into the variable B.
30	Checks to see if the "-1" has been read into the variable A. Since "-1" is the last datum in the list (see line 999), this checks for the end of the data list.
40	PRINTs the character with the value read into the variable B, at the screen position with the value read into the variable A.
50	Loops back to line 20 to read more data.
60 to 480	The data. The first item in every pair of numbers is the print location (to be read into the variable A), and the second item is the character code for that location (to be read into the variable B).
999	The data pair "-1,-1" which tells line 30 that this is the last line of

data. Two data items are needed here.
1000 A continuous loop which prevents the screen from scrolling.

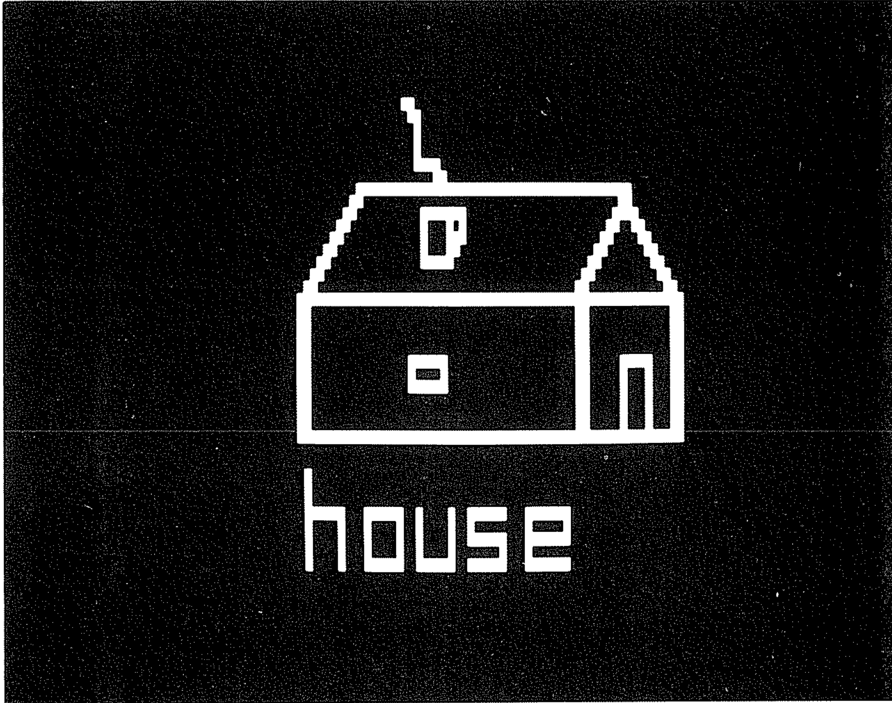


Photo 2-I

"But," you must be thinking, "that is a lot of work and a lot of programming just to draw a little house." You are right, of course. We will soon examine some techniques you can use to reduce the amount of work and program space it takes to program fast graphics.

BUILDING GRAPHICS IN STRING VARIABLES

If you plan to use a certain graphic on the screen several times, either simultaneously or at different times, you may want to build a string variable containing the graphic.

Before working with **graphics** characters in string variables, let's work a bit with **text** characters. The program in Listing 2-J builds the string variable B$ containing characters 89, 69, and 83. It then prints the string.

```
10 CLS
20 B$=CHR$(89)+CHR$(69)+CHR$(83)
30 PRINT B$
```

Listing 2-J

There is an easier way to do this, as you probably realize. Listing 2-K shows a program with the same result.

```
10 CLS
20 B$="YES"
30 PRINT B$
```

Listing 2-K

So what is the advantage of the program in Listing 2-J? If you are using text characters (as these programs do), there may be no advantage at all. But when you are using graphics characters, **you cannot type them in at the keyboard!** So you must use other means of storing them in string variables. Building them with the **CHR$** function is a good way. Listing 2-L shows a program which builds the string variable A$.

```
10 CLS
20 A$=CHR$(183)+CHR$(187)+CHR$(179)+CHR$(187)+
   CHR$(179)+CHR$(149)
30 PRINT A$
```

Listing 2-L

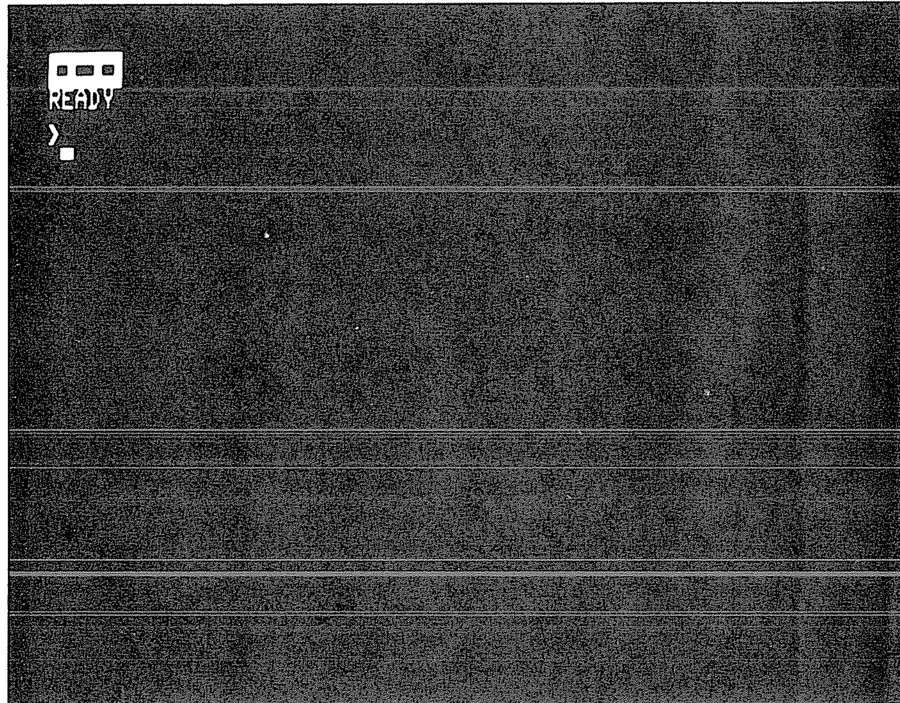


Photo 2-L

This program prints A$, which contains graphics characters that form a box. The program in Listing 2-M below uses the same characters to form a column of boxes. A line-by-line description follows the program.

```
10 CLS
20 A$=CHR$(183)+CHR$(187)+CHR$(179)+CHR$(187)+
   CHR$(179)+CHR$(149)
25 FOR J= 0 TO 14
30 PRINT @ J*64, A$;
35 NEXT J
40 GOTO 40
```

Listing 2-M

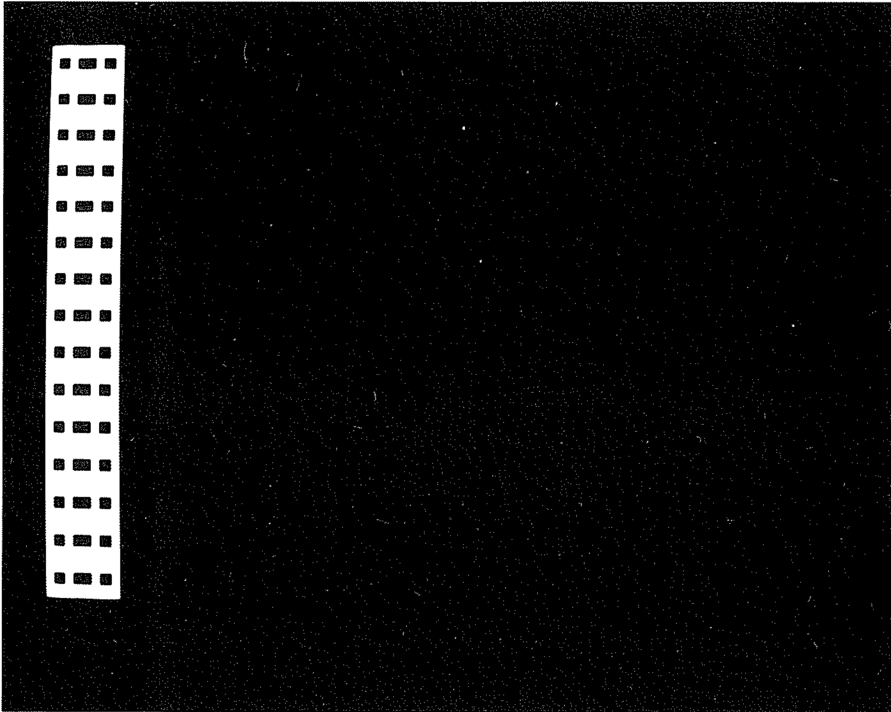


Photo 2-M

Line	Description
10	Clears the screen.
20	Builds the string variable A$ as characters 183, 187, 179, 187, 179, and 149.
25	Begins the FOR-NEXT loop from 0 to 14, with J as the looping variable.
30	PRINTs the string in A$. Each time through the loop, the value of J is increased by one, so the PRINT position changes by 64. This

moves the character down exactly one line (64 spaces) on the screen.
35 Ends the FOR-NEXT loop.
40 Loops back to line 40 until the BREAK key is pressed.

If you change line 30 to read as follows, the screen position will be incremented by 68, so the figure will appear one line down and four characters over each time. This creates a diagonal row of boxes.

```
30 PRINT @ J*68, A$;
```

Try a few other numbers to multiply with J in line 30. You will like the effects. When you consider the time it would take to draw these same figures with the SET instruction, you can see the advantage of PRINTing these graphics.

USING CONTROL CODES IN GRAPHIC STRINGS

The box created in the above programs was six graphics characters wide and one character tall. What if the graphic you want is more than one character tall? **You can use control characters to create graphics strings that cover more than one line on the screen.**

The program in Listing 2-N builds a string variable containing character 26. This character causes the graphics that follow it to be displayed on the next row down on the screen from the previous characters. Run the program, and you'll see what effect this has on the display.

```
10 CLS
20 C$=CHR$(191)+CHR$(26)+CHR$(191)+CHR$(26)+
   CHR$(191)
30 PRINT C$
```

Listing 2-N

Instead of displaying the graphic characters like this:

```
char char char
```

they are displayed like this:

```
char
     char
          char
```

Character 26 causes the each character to appear on the next line down. But what if you want the characters to appear directly under each other? Then you must add character 8, the backspace character. Change the program to look like this:

```
10 CLS
20 C$=CHR$(191)+CHR$(26)+CHR$(8)+CHR$(191)+
   CHR$(26)+CHR$(8)+CHR$(191)
30 PRINT C$
```

Listing 2-O

Before the second and third times that character 191 is displayed, the string displays a character 26 (move down) and a character 8 (move back). The graphic characters appear to be in a vertical row.

Photo 2-N

Let's use these control characters along with the graphic characters to build a happy face.

```
10 CLS
20 A$=CHR$(151)+CHR$(135)+CHR$(179)+CHR$(139)+CHR$(171)+
   CHR$(26)+CHR$(8)+CHR$(8)+CHR$(8)+CHR$(8)+CHR$(8)
   +CHR$(181)+CHR$(180)+CHR$(188)+CHR$(184)+CHR$(186)
30 PRINT @ 350, A$
40 GOTO 40
```

Listing 2-P

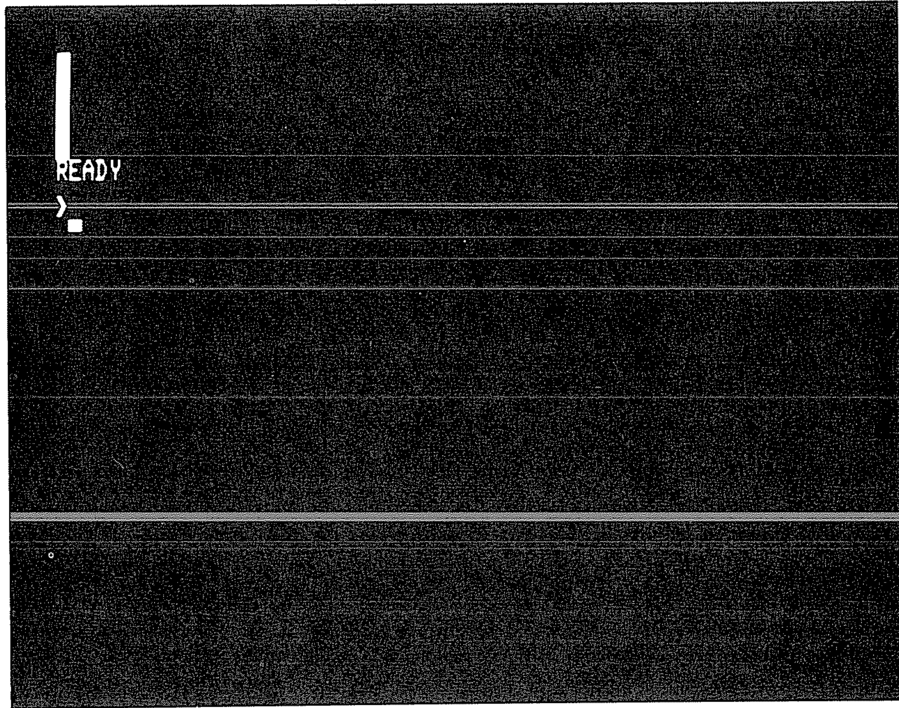


Photo 2-O

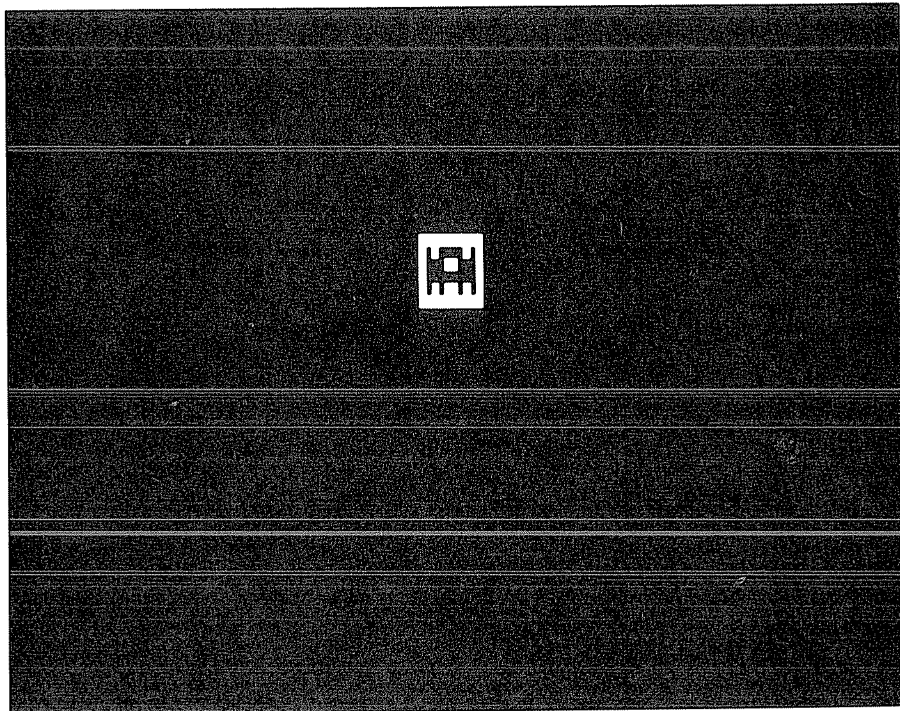


Photo 2-P

This program uses one character 26 to move down a line, then five of character 8 to move back five spaces. This lines up the beginning of the graphics on the second line with the graphics on the first line.

To be a bit more creative, try the program in Listing 2-Q. (So you won't have to retype them, lines 10 and 20 have not been changed from the program above.)

```
10 CLS
20 A$=CHR$(151)+CHR$(135)+CHR$(179)+CHR$(139)+CHR$(171)+
   CHR$(26)+CHR$(8)+CHR$(8)+CHR$(8)+CHR$(8)+CHR$(8)+
   CHR$(181)+CHR$(180)+CHR$(188)+CHR$(184)+CHR$(186)
30 PRINT @ RND (900), A$;
40 FOR J=1 TO 150: NEXT J
50 CLS
60 GOTO 30
```

Listing 2-Q

Here's a line-by-line description:

Line	Description
10	Clears the screen.
20	Builds the string variable B$ using graphics characters and the control characters 26 and 8.
30	Chooses a random number up to 900, and uses that number as the location to PRINT the face on the screen.
40	Pauses long enough for people to see the happy face.
50	Clears the screen again.
60	Returns to line 30 to PRINT the face again. You can press the BREAK key to escape this loop.

USING THE STRING$ FUNCTION TO BUILD GRAPHIC STRINGS

The CHR$ function allows you to add one character at to time to a string variable. In a similar way, the STRING$ function lets you add many chararacters at a time. Before examining the STRING$ function with graphic characters, let's use it with text characters.

```
10 CLS
20 PRINT STRING$(20,"A")
30 B$ = STRING$(30,66)
40 PRINT @ 320, B$;
50 GOTO 50
```

Listing 2-R

Can you predict the outcome of this program? RUN the program to see if your predictions were correct.

The program in Listing 2-R demonstrates five facts about the STRING$ function:

1) The first number in the parentheses tells how many characters will be used. This can be from 1 to 255.
2) If the second element in the parentheses is a character in quotation marks, a STRING$ of that character is created (as in line 20).
3) If the second element in the parentheses is a number without quotation marks, a STRING$ of **the character corresponding to that number** is created (see line 30).
4) A STRING$ of characters can be printed directly, as in line 20.
5) A STRING$ of characters can be assigned to a string variable (as in line 30) and PRINTed later (as in line 40).

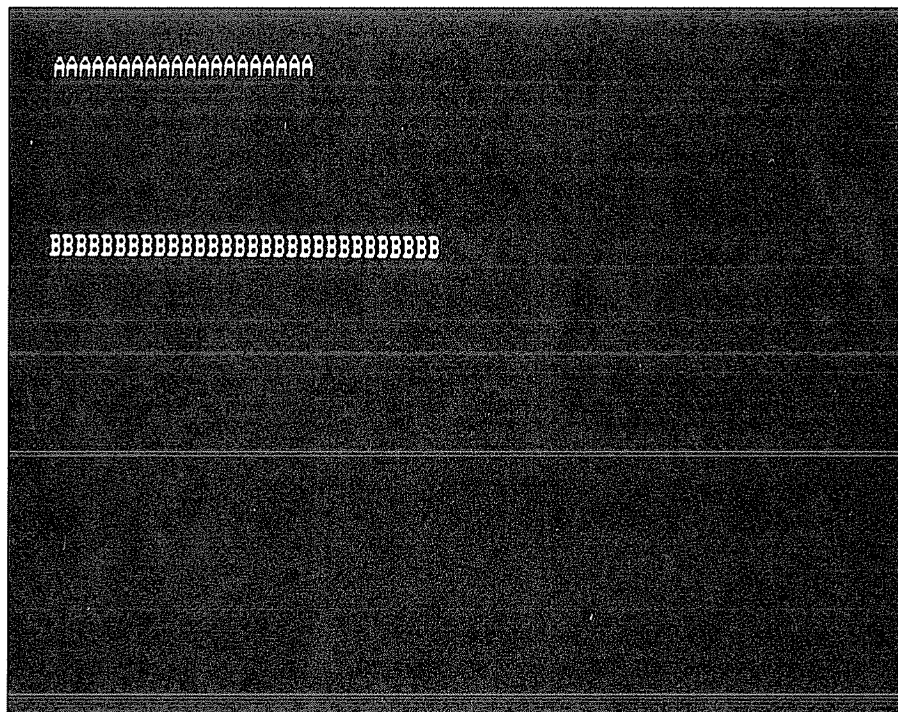


Photo 2-R

Using the STRING$ function with graphic characters works the same way. The only restriction is that the second element in the parentheses must be a number, since you can't type the graphics characters at the keyboard. Listing 2-S shows the same program with graphics characters instead.

```
10 CLS
20 PRINT STRING$(20,191)
```

```
30 B$ = STRING$(30,176)
40 PRINT @ 320, B$;
50 GOTO 50
```

Listing 2-S

Here is a line-by-line description:

Line	Description
10	Clears the screen.
20	Prints a STRING$ of 20 characters, each having code 191.
30	Assigns a STRING$ of 30 characters, each having code 176, to the string variable B$
40	Prints the string stored in variable B$ at screen location 320.
50	Loops back to line 50 to hold the display.

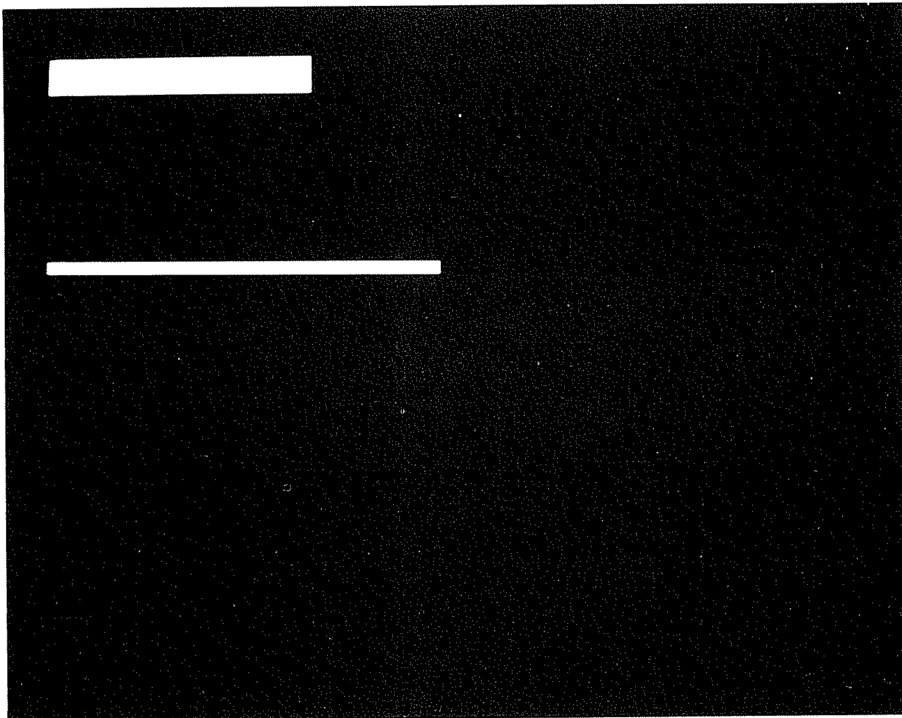


Photo 2-S

A PROBLEM SOLVED WITH THE STRING$ FUNCTION

One of the author's favorite graphics problems is, "What is the fastest way to fill the TRS-80 display with solid graphics characters in

BASIC?" The STRING$ function can help us solve that problem.

Return for a moment to the program in Listing 1-D in Chapter 1. This program fills the screen by SETting every pixel on the screen. But it is a painfully slow way to "clear the screen" with SET instead of RESET characters.

What if we PRINTed a character 191 (all white) character in every position of the screen? Listing 2-T shows a simple program that does that. It runs considerably faster than the program in Listing 1-D using the SET instruction. But there is one problem: after the character is PRINTed in the last position, the screen scrolls up one line.

```
10 CLS
20 FOR J=0 TO 1023
30 PRINT @ J, CHR$(191);
40 NEXT J
50 GOTO 50
```

Listing 2-T

It seems like a difficult problem: no matter what character is PRINTed in the last position on the screen, the screen will scroll. Listing 2-U presents a solution. Line 50 in this listing fills in the last character. Chapter 3 contains a complete explanation of how and why this works.

```
10 CLS
20 FOR J=0 TO 1022
30 PRINT @ J, CHR$(191);
40 NEXT J
50 POKE 16383, 191
60 GOTO 60
```

Listing 2-U

This program uses the **CHR$** function to PRINT a single character. How can we PRINT a STRING$? Listing 2-V shows the use of the STRING$ function to speed up the process. Since the program PRINTs a STRING$ of 32 characters in one instruction, it is faster yet.

```
10 CLS
20 A$ = STRING$(32,191)
30 FOR J=0 TO 31
40 PRINT @ J*32, A$;
50 NEXT J
60 GOTO 60
```

Listing 2-V

This program does what we want, but it PRINTS to the last screen position, so it causes a scroll. Let's make the program a little faster yet

before we fix the scrolling problem. Listing 2-W shows a program that builds a STRING$ of 64 characters instead of 32 characters. This should be faster.

```
10 CLS
20 A$ = STRING$ (64,191)
30 FOR J=0 TO 15
40 PRINT @ J*64, A$;
50 NEXT J
60 GOTO 60
```

Listing 2-W

When you RUN the program in progam in Listing 2-W above, another problem arises: an "Out of String Space" error. Let's take time out here for an explanation.

BASIC has two kinds of variables: numeric variables and string variables. The value for each numeric variable is stored in a special format in the memory of the computer, so the computer "knows" how much memory space to set aside for each numeric variable. But string variables can be as long as the programmer wants to make them, so the programmer must tell the computer how much space to set aside to store the strings. BASIC automatically sets aside enough space for 50 characters (bytes) of strings. If you want to use more than that, you must tell it to clear room for more characters. The CLEAR instruction tells the computer to set aside string space, and the number that follows it tells how many characters to allow room for. Line 5 in Listing 2-X CLEARs enough string space for 64 characters, so the STRING$ in line 20 doesn't cause the "Out of String Space" error.

```
5 CLEAR 64
10 CLS
20 A$ = STRING$ (64, 191)
30 FOR J=0 TO 15
40 PRINT @ J*64, A$;
50 NEXT J
60 GOTO 60
```

Listing 2-X

That solves the string space problem, but not the scrolling problem. Can you think of a way to avoid this scrolling? Listing 2-Y shows one solution. To avoid PRINTing to the last position, line 30 loops only 15 times, from 0 to 14. Line 54 PRINTs the bottom row of the screen, less the last character. And line 56 takes care of that last character. A line-by-line description follows the listing.

```
5 CLEAR 127
10 CLS
20 A$ = STRING$ (64,191)
```

```
30 FOR J=0 TO 14
40 PRINT @ J*64, A$;
50 NEXT J
54 PRINT @ 960, STRING$ (63,191)
56 POKE 16383, 191
60 GOTO 60
```

Listing 2-Y

Line	Description
5	Clears 127 bytes of string space.
10	Clears the screen.
20	Assigns a STRING$ of 64 characters having code 191 to the string variable A$
30	Begins a loop from 0 to 14 looping on the variable J
40	PRINTs the STRING$ each time through the loop. The first time through will be at position 0 (or 0*64). Another string will be printed after each 64 characters, or exactly one line down. The last one will be at position 896 (or 14*64), the beginning of the next-to-last line on the screen.
50	Ends loop J.
54	PRINTs a STRING$ of 63 characters having code 191 at 960, the beginning of the last line on the screen.
56	Creates the last character on the screen (more next chapter).
60	Loops continuously to inhibit the "Ready" prompt until the BREAK key is pressed.

MORE ON USING THE STRING$ FUNCTION

One more use of the STRING$ is with control characters. For example, instead of using the CHR$ function to add a character 8 each time we "back up" one space, we can use a STRING$ of character 8. Listing 2-Z shows a program using a STRING$ of character 8 as well as STRING$s of other characters.

```
10 CLS
20 CLEAR 200
30 A$ = CHR$(160) + CHR$(134) + CHR$(131) +
   CHR$(137) + STRING$(3,144) + CHR$(152) +
   STRING$(2,131) + CHR$(164) + CHR$(26) +
   STRING$(11,8) + CHR$(170) + STRING$(2,32) +
   CHR$(137) + CHR$(146) + CHR$(171) + CHR$(131) +
   CHR$(152) + CHR$(129) + CHR$(32) + CHR$(170)
40 A$ = A$ + CHR$(26) + STRING$(11,8) + CHR$(130)
   + CHR$(139) + CHR$(172) + CHR$(176) + CHR$(184)
   + CHR$(142) + CHR$(172) + CHR$(176) + CHR$(184)
   + CHR$(142) + CHR$(131)
50 PRINT A$;
60 PRINT @ 340, A$;
```

Listing 2-Z

Photo 2-Z

This program starts building a STRING$ is line 30 and finishes it in line 40. Notice the "STRING$(11,8)" in line 30 which moves the cursor back eleven spaces.

Do you wonder if the author spent untold hours drawing graphic characters out on a paper to design the graphics shown in some of these graphics programs? In the next chapter you will see some of the software tools you can use to make similar complex graphics with relative ease.

Chapter Summary

The TRS-80 screen has 1024 screen positions numbered from left to right and top to bottom. These are sometimes called the "PRINT AT" positions. They begin with 0 in the upper left corner and end with 1023 in the lower right corner.

Programs can PRINT at any of these locations using the "PRINT AT" instruction in this format: **PRINT @ a,b** where a is a screen position from 0 to 1023 and b is a character number between 1 and 255.

Simple graphics can be created by PRINTing regular text characters on the screen using the PRINT @ instruction.

Pictures using graphic characters can be PRINTed on the screen using the CHR$ function with the PRINT @ instruction.

The codes for graphic characters range from 128 to 191.

READ and DATA instructions can be used to print many graphic characters in many different positions with only one PRINT @ instruction.

Graphic characters can be assigned to string variables so they can be printed quickly and/or repeatedly. (Example: B$ = CHR$(189) + CHR$(188))

Control codes can be used to control the relative positions of characters within a string. Character 26 moves the position down one character, and character 8 moves the position left one character.

The STRING$ function can be used with text, graphic, and control characters to reduce the number of program instructions needed to build and print a graphic string.

The STRING$ function uses string space in a BASIC program, and sometimes more string space must be cleared using a CLEAR instruction.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Write a program that draws a square using the PRINT instruction with an "*".
B. Write a program that prints a box on the TRS-80 screen using PRINT @ and graphics characters.
C. Write a program that prints a box on the TRS-80 screen by assigning graphics characters to a string variable and printing the variable.
D. Write a program using the STRING$ function that fills one line on the TRS-80 screen with a row of graphics characters.
E. Write a program that draws a heart on the screen by assigning graphics characters to a string variable and printing the variable. Use Character 26 and a STRING$ of Character 8 to go to the next line of the screen.

More Programming Practice

1. Write a program that draws a horse on the screen using PRINT @ instructions and regular text characters.

2. Write a program that draws a flower on the screen using the PRINT @ instruction with graphic characters.

3. Build a string variable that forms the shape of each of your initials. PRINT the variables on the screen so your initials are displayed simultaneously.

4. Write a program that assigns graphics characters and control codes 26 and 8 to a string variable, so that three characters are printed on the screen in a vertical column.

5. Since there are 64 graphics characters and 1024 screen positions, you can fill the screen by PRINTing 1024/64 or 16 each of the graphics characters. Write a program that PRINTs STRING$s of 16 each of the graphics characters, beginning with 128 and going to 191, on the screen at once. These characters should fill the screen. Prevent the screen from scrolling.

6. When you type PRINT MEM, the computer prints the number of bytes (characters) of memory available for BASIC programming. Does CLEARing string space change the amount of memory available for programming? Write a program that tests the amount of memory available before and after CLEARing 500 bytes of memory.

Chapter Checkup

1. Listing 2-F contains a two-line program. Why isn't a line like this:
   30 GOTO 30
   needed in the program?

2. Listing 2-H contains the following program:

```
10 CLS
20 PRINT @ 479, CHR$(170);
30 PRINT @ 480, CHR$(131);
40 PRINT @ 481, CHR$(131);
50 PRINT @ 482, CHR$(149);
60 PRINT @ 543, CHR$(130);
70 PRINT @ 544, CHR$(131);
80 PRINT @ 545, CHR$(131);
90 PRINT @ 546, CHR$(129);
100 GOTO 100
```

   Lines 20 to 50 can be combined into one line like this:
   **20 PRINT @ 479, CHR$(170);CHR$(131);
   CHR$(131);CHR$(149);**
   a) What are two advantages of the change?
   b) Lines 60 to 90 can be combined in a similar way. Write the new line 60 that contains this change.

3. Line 999 of Listing 2-I contains these two data items: "-1,-1." Why are two items needed here instead of just one?

4. Line 30 of Listing 2-Q uses **RND(900)** to choose the screen location for PRINTing the graphic. The screen locations go up to 1023. Why do you think **RND(900)** was selected instead of **RND(1023)**?

5. Line 5 of Listing 2-X CLEARs 64 bytes of string space, but Line 5 of Listing 2-Y CLEARs 127 bytes. Why were these extra bytes of string space needed?

6. The programs in Listings 1-D and 2-Y both fill the screen completely with graphics. Using the second hand or second counter on a watch, time each of these programs as it runs. Divide the longer time by the shorter time. How many times as fast is the PRINT version compared to the SET version?

7. What do you predict as the outcome of the following program? Run the program to see if your prediction proved correct.

```
10 CLS
20 A$ = STRING$(5,191) + CHR$(26) + STRING$(5,32) + CHR$(26) +
   STRING$(5,191)
30 PRINT A$
```

8. When turn on the TRS-80 Model I or Model III to begin programming in BASIC, how many bytes of string space are available?

9. If you type **PRINT FRE(" ")**, the TRS-80 will print the number of free bytes of string space. When you first turn on your TRS-80, check the number of free bytes of string space. Run a program that CLEARs more bytes. Check it again. Then type **NEW** and press **ENTER**. Does removing the program from memory with the **NEW** instruction change the amount of string space CLEARed?

10. What is the result of the following program? Explain what happened.

```
10 CLS
20 PRINT STRING$(5,"176")
```

Chapter 3.
Introduction to
TRS-80 Graphics Techniques:
POKE Graphics and the Video Memory

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write the locations of the beginning and end of the video memory on the TRS-80 Model I, Model III and Model 4.
B. Write an example (in proper format) of a POKE instruction which places a graphics character on the display.
C. Write a program that draws a vertical line on the display, using POKE graphics.
D. Write an example of a simple program with an INKEY$ instruction that POKEs values corresponding to keystrokes, to the video RAM.
E. Write a program that PEEKs each position of the video RAM and displays each value.
F. Write a program that hides a string of characters in the odd-numbered locations of the video memory while a RUN instruction is executed.

INTRODUCTION

Thus far we have examined two ways to produce graphics on a TRS-80 Model I, Model III, or Model 4 screen: SETting points and PRINTing graphics characters. The third and final major way to accomplish graphics programming is by using the POKE instruction.

The POKE instruction is aptly named: it places or "pokes" a certain value into a specified memory location. The TRS-80 microcomputers, like all microcomputers, have memory in which to store information. The POKE instruction simply lets the programmer access this memory directly. For example, this instruction:

```
POKE 25020, 54
```

places the value 54 into memory location 25020.

What makes this information useful to the graphics programmer is that part of the TRS-80 memory determines what is displayed on the TRS-80 screen. This part of memory is commonly called "video memory" or "video RAM." By POKEing values to the video memory, we can control what appears on the computer's screen.

The video memory resides in locations 15360 to 16383 inclusive. The video memory values correspond to the screen locations in much the same way as the PRINT @ locations. Location 15360 is in the upper left corner, and value 16383 is in the lower right. The location on the screen of any particular video memory location can be determined by adding

15360 to the PRINT @ location.

POKEING VALUES INTO THE VIDEO MEMORY

Consider the program in Listing 3-A. This program POKEs the value 191 into memory location 16000. Since 16000 is between 15360 and 16383, it is in the video memory. When you run this program, character 191 appears at PRINT @ location 640. The number 640 can be found by subtracting 16000 - 15360.

```
10 CLS
20 POKE 16000, 191
```

Listing 3-A

The program in Listing 3-B places the value 191 in the lowest and highest locations of the video memory. Where do you think these characters will appear on the screen?

```
10 CLS
20 POKE 15360, 191
30 POKE 16383, 191
40 PRINT @ 128, "They are in two corners."
```

Listing 3-B

What would happen if we POKEd the value 191 into ALL the locations in the video RAM? Before you look at Listing 3-C, can you think of the most efficient way to program this multiple POKE?

```
10 CLS
20 FOR J=15360 TO 16383
30 POKE J, 191
40 NEXT J
50 GOTO 50
```

Listing 3-C

The FOR-NEXT loop in lines 20 to 40 POKE the value 191 into every memory location between 15360 and 16383. This fills every screen location with character 191. This program shows another way to "clear the screen" with all "SET" characters, as was discussed in Chapter 2.

Do you recall the unexplained POKE instruction in those screen-filling programs in Chapter 2? It was the special POKE that allowed us to fill that last location of the screen without causing the screen to scroll. The programs contained the following instruction:

```
POKE 16383, 191
```

Does this instruction make sense to you now?

DRAWING LINES

Drawing lines with the POKE instruction has limitations similar to those of the PRINT @ instruction. Single pixels cannot easily be turned on or off, as they can in the SET/RESET instruction. But within that restriction, lines can be drawn anywhere on the screen.

To draw a line, we may first choose a graphics character to use to build the line. Then we may decide where on the screen we want the line, which will determine which video memory locations we use to display it.

To draw a line on the top row of the screen, we may decide to use character 140, which "turns on" the two pixels in the vertical middle of the character (see Appendix 3). To determine the memory locations, we may begin with the PRINT @ locations, which are 0 to 63 on the top row of the screen (see Appendix 1). By adding 15360 to each of these numbers, we arrive at 15360 and 15423 as our beginning and ending loop values. So our program will look like this:

```
10 CLS
20 FOR J = 15360 TO 15423
30 POKE J, 140
40 NEXTJ
```

Listing 3-E

When you run this program, you will notice that BASIC's "Ready" prompt overwrites the line that was drawn. This is because the program has POKEd the values to the video memory and thus has not changed the current PRINT location since clearing the screen in line 10. So the "Ready" prompt appears at the current PRINT location, which is still the top left corner of the screen.

We could avoid this by changing the PRINT location before ending the program, like this:

```
50 PRINT @ 320, "";
```

Or we could end the program with an endless loop, like this:

```
50 GOTO 50
```

Drawing vertical lines is similar. Instead of POKEing the value of the graphics character into adjacent memory locations in a horizontal row, we POKE them into adjacent locations in a vertical row.

Listing 3-F shows an example. This time the character used is 149, which "turns on" the pixels on the left half of the graphics character. The loop starts again at 15360, the top left corner of the screen. But this time it ends at 16320, which corresponds to PRINT @ location 960, the

lower left corner of the screen. The loop STEPs by 64, which means that every 64th memory location will be POKEd with character 149. Since there are 64 characters per line, the effect is to display each character below the previous one, thus drawing the vertical line. Line 50 is added to inhibit the "Ready" prompt until the BREAK key is pressed.

```
10 CLS
20 FOR J = 15360 TO 16320 STEP 64
30 POKE J, 149
40 NEXT J
50 GOTO 50
```

Listing 3-F

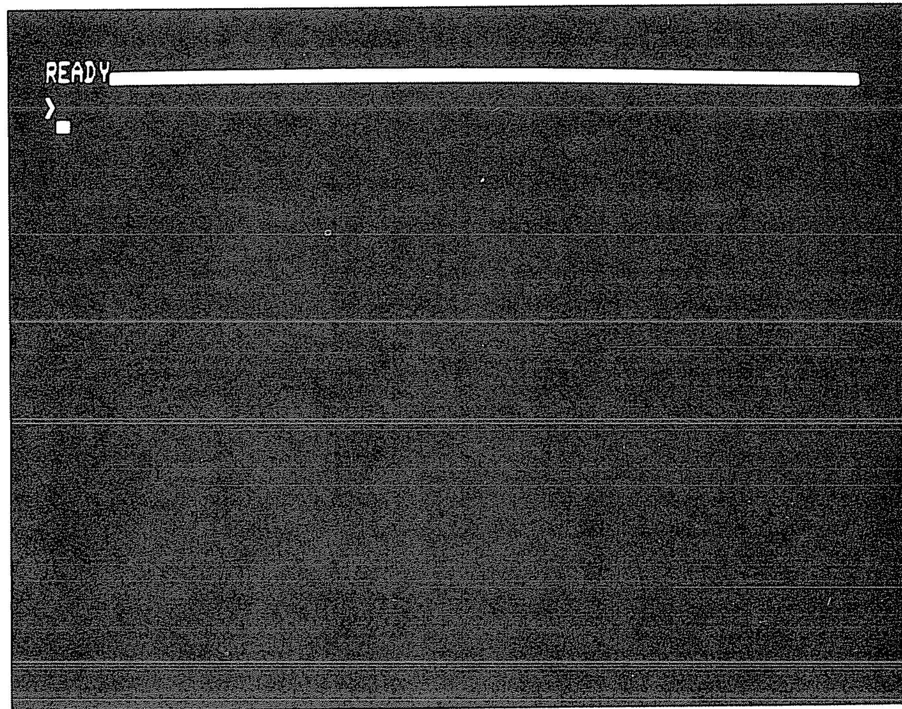


Photo 3-E

MORE COMPLEX PATTERNS

The program in Listing 2-I showed a picture of a house that was drawn by PRINTing various characters at various screen locations. The program looked like this:

```
10 CLS
20 READ A,B
```

```
30 IF A= -1 THEN 1000
40 PRINT @ A, CHR$(B);
50 GOTO 20
    -
    -
(Lots of DATA statements)
    -
    -
999 DATA -1, -1
1000 GOTO 1000
```

Listing 3-D

The PRINT @ instruction in line 30 PRINTs the characters at the specified locations. Can you change line 30 to use a POKE instruction instead of the PRINT @? Can the same DATA values be used? The values to be displayed (contained in the variable B) would not change. But how would the memory location compare to the PRINT @ location?

Of course! We must add 15360 to the PRINT @ location to find the proper POKE location. So the new line 30 would read as follows:

```
30 POKE A+15360, B
```

This entire house picture can be created without SETting a single point or PRINTing a even one graphics character!

THE NEED FOR A BETTER KEYBOARD INPUT ROUTINE

We are going to take a close look at a specialized program that uses the video memory and controls what kinds of characters can be typed in at the keyboard. But first let's look at the reason such a routine is needed.

The program in Listing 3-E shows an example of keyboard input.

```
10 CLS
20 INPUT "WHAT IS YOUR NAME"; N$
30 INPUT "WHAT IS THE FIRST NUMBER";N1
40 INPUT "WHAT IS THE SECOND NUMBER";N2
50 PRINT N$ "," N1 "+" N2 "=" N1+N2
```

Listing 3-E

Here is a line-by-line description of the program:

Line	Description
10	Clears the screen.
20	Displays the question on the screen, accepts a keyboard response,

and assigns the keyboard input to the string variable N$.

30 Displays the question on the screen, accepts a keyboard response, and assigns the keyboard input to the numeric variable N1.

40 Displays the question on the screen, accepts a keyboard response, and assigns the keyboard input to the numeric variable N2.

50 Displays the name (N$), a comma, the first number (N1), a plus sign, the second number (N2), an equals sign, and the sum of the first number and the second number (N1 + N2).

Keyboard input in this program in accomplished with the INPUT instruction. This instruction is the easiest and most direct way to accept keyboard input.

But it has these limitations:

1. The program cannot control how long a keyboard response will be. For example, an mailing list program might list names in columns and allow twelve spaces for each name. If the name is more than twelve characters long, the name must be chopped off or the format of the columns will be disrupted. Controlling the length of the input would eliminate this problem.
2. The programmer cannot control which keystrokes are accepted. If a numeric variable is used in the INPUT statement (as in lines 30 and 40 in Listing 3-E), typing something that begins with a letter will produce a "Redo" error message, again possibly messing up the format of the screen. This could be avoided if the keyboard input could be limited to numbers only.
3. If the CLEAR key is pressed during the keyboard input, the entire screen in cleared. That could really destroy an attractive screen.

These limitations demonstrate the need for another way to accept keyboard input.

A BETTER KEYBOARD INPUT ROUTINE

Listing 3-F shows the beginning of a better keyboard input routine. (We will add to this program little by little, so the line numbers are not all there yet.) The line-by-line descriptions detail the unique features of this routine.

```
20 LO = PEEK(16416) + PEEK(16417) * 256
40 I$ = INKEY$
100 I$ = INKEY$
110 IF I$="" THEN
      GOTO 100
140 I = ASC(I$)
190 POKE LO,I
210 LO = LO + 1
230 GOTO 100
```

Listing 3-F

Line	Description
20	The value "PEEK(16416) + PEEK(16417) * 256" defines the current cursor location in the video memory. For example, when the screen is cleared with a "CLS" instruction, the value returned would be 15360, the top left position of the video memory. No matter what was last PRINTed to the screen, this value specifies the location where the **next** printing would occur by default. This value is assigned to the variable LO, which will contain the current screen location throughout this routine.
40	The keyboard is scanned. If a key has been pressed since the last INPUT or INKEY$ instruction, it will be stored in the the variable I$. This first INKEY$ instruction picks up any previous, unintended keystrokes.
100	This INKEY$ instruction also scans the keyboard, and any keystroke pressed is stored in the variable I$, the new value replacing any previous value in I$. For example, if the user presses the "A" key, I$ contains "A"; but if no key is pressed, I$ contains "" (null).
110	If no key is pressed and I$ therefore contains null, the program loops back to line 100 to scan for another character. The program will loop back and forth between lines 100 and 110 until a key is pressed. At that point, the program proceeds to line 140.
140	The number assigned to the string I$ (its ASCII value) is stored in the numeric variable I. (Appendix 2 contains a list of all the ASCII characters used in the TRS-80 Model I, III, and 4.) For example, if the "A" key is pressed, the variable I would contain 65, the ASCII value of "A."
190	The ASCII value of the keyboard input is POKEd into the current cursor location. This displays the character corresponding with the depressed key at the current cursor location. In other words, what you pressed is what you see.
210	The variable containing the current location (LO) is incremented, so the next character will be POKEd into the next position on the screen.
230	The program returns to line 100 to wait for the next keystroke.

The total effect of this routine is described in line 190: the character corresponding to each key you press appears on the screen.

When you first run this program, it looks like nothing is happening. But when you press a key, you will see the effect. The position of the last letter displayed tells where the next position is. Wouldn't it be nice to have a cursor show where the next screen position would be?

CREATING A BLINKING CURSOR AND STORING THE KEYSTROKES PRESSED

Listing 3-G shows lines that can be added to the previous routine to create a blinking cursor. (Lines marked with an asterisk appeared in

the original routine, but have been changed.)

```
    60 CU = 32
    70 CU = 168 - CU
    80 POKE LO, CU
    90 FOR J = 1 TO 20
* 110 IF I$ = "" THEN
        NEXT J:
        GOTO 70
   120 J = 20
   130 NEXT J
   220 CU = 32
* 230 GOTO 70
```

Listing 3-G

Here is a line-by-line description of these added lines:

Line	Description
60	The variable CU contains the value of the cursor character to be POKEd onto the screen. In other words, the character corresponding to the value of CU is shown as the current cursor character. The value starts as 32, a space.
70	This is a tricky one. The value of CU is subtracted from 168, and that new value is assigned to CU. CU begins as 32, then is immediately assigned 168-32, or 136. Since CU is the cursor character, a new cursor character is assigned by this statement. The program loops back to this line repeatedly from lines 110 and 230. The second time this line is executed, CU is assigned a value of 168-136, or 32. Thus the effect of this line is to cause the cursor character to alternate between 32 (a space) and 136 (a graphics character). This creates the blinking cursor.
80	The current value of the cursor character is POKEd into the current cursor location. This causes the cursor character to appear on the screen.
90	This line begins a loop with J as the looping variable.
110	If I$ is null, no key has been pressed. When this happens, the "NEXT J" loops back to line 90 for another keyboard scan (line 100). After the FOR-NEXT loop is finished, this line returns to line 70 to switch the cursor character. So the loop beginning in line 90 controls the wait before the cursor character is switched.
120	If a keystroke is received, the program goes to this line. J is set at 20 so the FOR-NEXT loop can be ended in line 130.
130	The FOR-NEXT loop is ended. The previous step and this one demonstrate a good programming practice: terminate each FOR-NEXT loop completely before proceeding with the program.
220	The cursor character variable (CU) is assigned a value of 32. When the program returns to line 70, this will be subtracted from 168 to yield a value of 136. This line causes the graphics character to appear as an almost constant cursor as long as the user types

continuously, then blink when the user pauses.
230 Program control is returned to line 70 for another keystroke.

For this routine to be useful, it must somehow keep track of which keys were pressed. Listing 3-H shows how this is accomplished.

```
50 S$ = ""
200 S$ = S$ + I$
```

Listing 3-H

Line	Description
50	S$ is the string variable that will store the keys pressed. At the beginning of the routine, S$ is set to "" or null.
200	Immediately after the value of the key pressed is POKEd to the screen, the character corresponding to the key pressed (I$) is added to S$, building the string.

So far our routine shows a blinking cursor, displays the character of each key pressed to the screen, and builds a string of the keys pressed. To see the value of the string being built,

A) Run the program.
B) Press the BREAK key to stop the program.
C) Type **PRINT A$ <ENTER>** to display the value of A$.

ENABLING THE LEFT ARROW

What happens when you press the left arrow key to erase the last character you typed? In the routine above, the number corresponding to the left arrow (8) is POKEd into the video memory with the rest of the characters. The code in Listing 3-I makes it erase the last character instead.

```
150 IF I = 8 AND S$ <> "" THEN
       POKE LO, 32:
       LO = LO - 1:
       S$ = LEFT$ (S$, LEN (S$)-1):
       GOTO 70
```

Listing 3-I

This line first checks to see if the back arrow is pressed (I = 8) and other keys have already been pressed (S$ <> ""). If these two conditions are met, the following steps are executed:

A) 32 (a space) is POKEd into the current cursor location. This assures that the other cursor character (136) is not left behind.
B) The current cursor location (LO) is decremented by 1. This moves the location one character to the left.
C) The right-most character of S$ is deleted.

D) The routine returns to line 70 for another character. This prevents the lines beginning at 190 from being executed, and thus prevents the 8 from being POKEd to the screen.

MAKING THE ROUTINE A SUBROUTINE

This routine is becoming quite lengthy, so you would not want to include it in your program each time you needed to accept input from the keyboard. So that it will have to be included only once, we can make it into a subroutine to be called as needed. Listing 3-J changes our routine into a subroutine.

```
10 GOTO 500
160 IF I = 13 THEN
      POKE LO, 32:
      PRINT:
      RETURN
500 CLS: PRINT "What is your name? ";
540 GOSUB 20
```

Listing 3-J

Line	Description
10	The program "jumps around" the input subroutine to the main program. Keeping the subroutine near the beginning of the program helps it respond to keystrokes more quickly.
160	After accepting a keystroke, storing its ASCII value, and checking to be sure it isn't a backspace, the subroutine checks for the ENTER key (character 13). This key is used to terminate the keyboard input, so it ends the subroutine. When character 13 is detected, the subroutine does the following: A) POKEs 32 (a space) into the current cursor location to be sure no trailing cursor character is left behind. B) Executes one PRINT statement. Since the routine POKEs to the video memory rather than PRINTing to the screen, the print location has not changed since the subroutine began. This PRINT statement moves the new print location to the next line on the screen, and thus guards against printing over the characters accepted in the subroutine. C) RETURNs from the subroutine to the main program.
500	This line begins the main body of the program. It clears the screen, then prints a question. Notice that the question has a question mark. An INPUT statement automatically displays a question mark after each question, but our subroutine doesn't. Also, notice that the semicolon after the question causes the answer to be displayed on the same line as the question.
540	This line calls the subroutine. In this program, "GOSUB 20" is approximately equivalent to "INPUT S$."

CHECKING THE INPUT

One of the limitations of the INPUT statement was that the length of the string and the specific characters accepted could not be controlled. Statements to control these are the next additions to our subroutine.

The lines in Listing 3-K add the capability of checking the length of the string being built while awaiting more keystrokes.

```
30 L1 = LO
180 IF LO - L1 = LE THEN
      GOTO 70
530 LE = 12
```

Listing 3-K

Line	Description
30	The numeric variable L1 stores the initial cursor location. In this line, which appears near the beginning of the subroutine, L1 is assigned its value.
180	LE is the maximum length of the string to be accepted (see line 530). If the current location (LO) minus the beginning location (L1) equals the maximum length of the string (LE), the string is as long as is allowed. The program loops back to line 70 for another keystroke rather than continuing to build the string.
530	This line is located in the main program and precedes the call to the subroutine. By assigning the value of LE, the main program specifies the maximum length of the string to be built in the subroutine.

The last addition to our subroutine checks which characters are being pressed.

```
170    IF I<LV OR I>HV THEN
             GOTO 70
510 LV = 32
520 HV = 90
```

Listing 3-L

Line	Description
170	The numeric variable LV contains the lowest ASCII value which will be accepted in the subroutine, and HV contains the highest value. If the variable I contains a value less than the low value or greater than the high value, the program loops back to line 70 to await another keystroke.
510	This line and the next are in the main body of the program and precedes the call to the subroutine. In it the low value is set at 32, the ASCII value for a space.

520 The high value is set at 90, the value for a "Z." Characters greater than "Z," including all lower-case letters, will not be accepted.

FINISHING THE MAIN PROGRAM

Now that our keyboard input subroutine is complete, we can change the main body of the program to make full use of the subroutine.

```
550 N$ = S$
560 PRINT "What is the first number? ";
570 LV = 48
580 HV = 57
590 LE = 3
600 GOSUB 20
610 N1 = VAL (S$)
620 PRINT "What is the second number? ";
630 GOSUB 20
640 N2 = VAL (S$)
650 PRINT N$; ","; N1; "+"; N2; "="; N1+N2
```

Listing 3-M

Line	Description
550	The value of S$ (which is currently the person's name) is assigned to the variable N$. Since S$ is assigned a new value every time the routine is called, its value must be used or transferred to another variable before the subroutine is called again.
560	This line PRINTs the question on the screen. Since the question ends with a semicolon, the line feed is suppressed and the current cursor position remains on the same line of the display.
570	The lowest ASCII value of keystrokes accepted is set at 48, which corresponds to a "0."
580	The highest ASCII value of keystrokes accepted is set at 57, which corresponds to a "9."
590	The maximum length of the string to be accepted is set at 3.
600	The keyboard input subroutine is called.
610	The value of S$, the string variable assigned in the subroutine, is assigned to the variable N1.
620	The second question is PRINTed.
630	The subroutine is called again. Notice that since the variables LV, HV, and LE were not assigned new values, they retain their values from the previous call of the subroutine.
640	The value of S$ is assigned to the variable N2.
650	The person's name, the first number, the second number, and the sum of the two numbers are PRINTed on the screen with appropriate punctuation.

So the whole program should look like this:

```
10 GOTO 500
20 LO = PEEK(16416) + PEEK(16417) * 256
30 L1 = LO
40 I$ = INKEY$
50 S$ = ""
60 CU = 32
70 CU = 168 - CU
80 POKE LO, CU
90 FOR J = 1 TO 20
100 I$ = INKEY$
110 IF I$ = "" THEN
    NEXT J:
    GOTO 70
120 J = 20
130 NEXT J
140 I = ASC(I$)
150 IF I = 8 AND S$ <> "" THEN
    POKE LO, 32:
    LO = LO - 1:
    S$ = LEFT$ (S$, LEN (S$)-1):
    GOTO 70
160 IF I = 13 THEN
    POKE LO, 32:
    PRINT:
    RETURN
170    IF I<LV OR I>HV THEN GOTO 70
180 IF LO - L1 = LE THEN GOTO 70
190 POKE LO,I
200 S$ = S$ + I$
210 LO = LO + 1
220 CU = 32
230 GOTO 70
500 CLS: PRINT "What is your name? ";
510 LV = 32
520 HV = 90
530 LE = 12
540 GOSUB 20
550 N$ = S$
560 PRINT "What is the first number? ";
570 LV = 48
580 HV = 57
590 LE = 3
600 GOSUB 20
610 N1 = VAL (S$)
620 PRINT "What is the second number? ";
630 GOSUB 20
640 N2 = VAL (S$)
650 PRINT N$; ","; N1; "+"; N2; "="; N1+N2
```

Listing 3-N

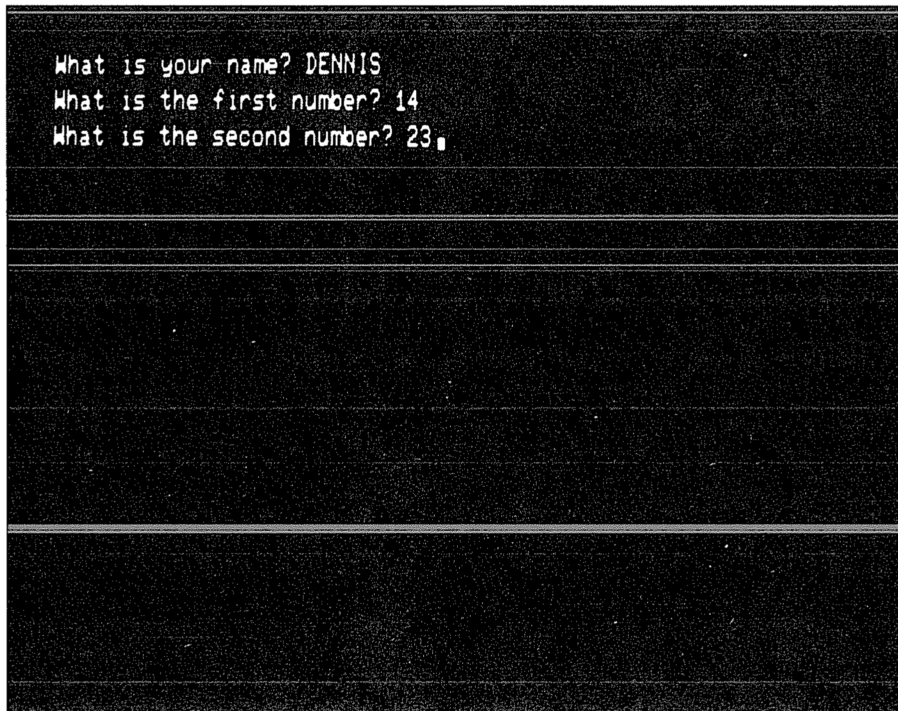


Photo 3-N

Lines 20 to 230 can be used as a subroutine in any program. In fact, they can be compressed into 3 lines, as follows:

```
10 LO = PEEK (16416) + PEEK (16417) * 256:
       L1 = LO:
       I$ = INKEY$:
       S$ = "":
       CU = 32
20 CU = 168 - CU:
       POKE LO, CU:
       FOR J = 1 TO 20:
       I$ = INKEY$:
       IF I$ = "" THEN
            NEXT J: GOTO 20 ELSE
       J = 20:
       NEXT J:
       I = ASC (I$):
       IF I = 8 AND S$ <> "" THEN
            POKE LO, 32:
            LO = LO - 1:
            S$ = LEFT$ (S$, LEN (S$) -1):
            GOTO 20
```

```
30 IF I = 13 THEN
        POKE LO, 32:
        PRINT:
        RETURN ELSE
     IF I < LV OR I > HV THEN
        GOTO 20 ELSE
     IF LO - L1 = LE THEN
        GOTO 20 ELSE
     POKE LO, I:
     S$ = S$ + I$:
     LO = LO + 1:
     CU = 32:
     GOTO 20
```

Listing 3-O

In using this program, all the spaces may be deleted, except those between "LE" and "THEN" shown in bold letters in line 30. The TRS-80 processor may interpret "LETHEN" as "LET HEN," which will produce a syntax error in this line.

So this subroutine uses the video memory and the INKEY$ instruction to improve the control the programmer has over keyboard input.

READING GRAPHICS FROM THE VIDEO MEMORY

We have seen some complex pictures drawn on our TRS-80 display. The author did not spend hours calculating the graphics characters and locations for these pictures. Rather, he invested his time in writing a program that would do the hard work for him.

The program has three main parts: a drawing section, an error-trapping routine, and a screen-reading program. This program combines SET/RESET graphics (used to create the picture) with the use of the video memory to read the graphics from the screen.

First let's look at the drawing section.

```
20 CLS
30 I$ = INKEY$
40 PRINT 960, X; Y;
50 I$ = INKEY$: IF I$ = "" THEN 50
60 IF I$ = CHR$ (9) THEN
     X = X + 1:
     SET (X,Y):
     GOTO 40 '*** CHR$ (9) = RIGHT ARROW ***
70 IF I$ = CHR$ (8) THEN
     X = X - 1:
     SET (X,Y):
     GOTO 40 '*** CHR$ (8) = LEFT ARROW ***
```

```
80 IF I$ = CHR$ (91) THEN
      Y = Y - 1:
      SET (X,Y):
      GOTO 40 '*** CHR$ (91) = UP ARROW ***
90 IF I$ = CHR$ (10) THEN
      Y = Y + 1:
      SET (X,Y):
      GOTO 40 '*** CHR$ (10) = DOWN ARROW ***
100 IF I$ = CHR$ (32) THEN
      RESET (X,Y):
      GOTO 40 '*** CHR$ (32) = SPACE ***
120 GOTO 40
```

Listing 3-P

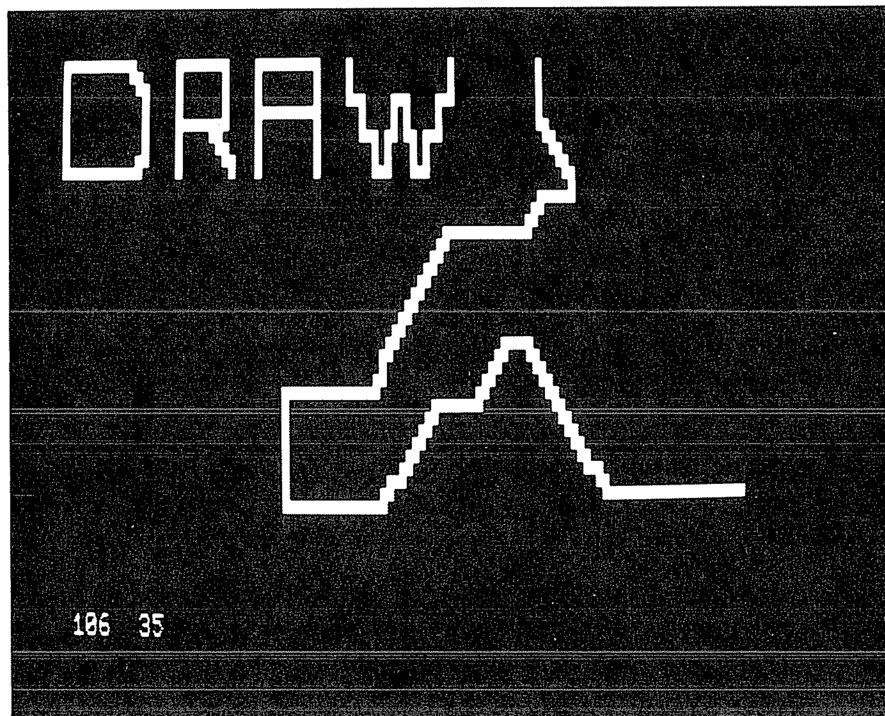


Photo 3-P

Remarks have been added to this listing to minimize the extra explanation needed. When one of the arrow keys is pressed, the program SETs a pixel. When the space bar is pressed, the program RESETs the pixel. The current values of X and Y are PRINTed at location 960.

Listing 3-Q shows the error-trapping routine. When an error occurs, the program jumps to line 130 to PRINT a message, pause, PRINT spaces

over the message, and RESUME at the program line after the error line.

```
10 ON ERROR GOTO 130
130 PRINT @ 960, "ERROR...";
140 FOR K = 1 TO 400: NEXT K
150 PRINT @ 960, "        ";
160 RESUME NEXT
```

Listing 3-Q

The remainder of the program (Listing 3-R) reads the video ram to determine which graphics characters are at which location. If the program finds a regular space (character 32) or a graphics space produced by a RESET instruction (character 128), it goes to the next location. Otherwise, it PRINTs the location and the value contained at that location. It awaits a keystroke before proceeding to allow the user time to write down the numbers.

```
110 IF I$ = "Y" OR I$ = "y" THEN
          GOTO 170
170 FOR J = 15360 TO 16383
180 IF PEEK (J) = 32 OR PEEK (J) = 128
          THEN NEXT J: END
190 PRINT @ 960, J; PEEK (J);
200 I$ = INKEY$: IF I$ = "" THEN 200
210 PRINT @ 960, "LOOKING.....";
220 NEXT J
```

Listing 3-R

To use this program, press the arrow keys to draw. To move without drawing, press the arrow key, then the space bar for each step. When you are ready to read your graphic from the screen, press the "Y" key. As you write down the numbers, press any key (except BREAK!) to go to the next set of numbers.

As you can see in this program, **even when you use SET/RESET (or PRINT) graphics, they can be read directly from the video memory.**

STORING NUMBERS IN THE VIDEO MEMORY

A certain feature of the video memory makes is useful in storing numbers.

When a BASIC program is first RUN, all the variables are CLEARed; that is, all the numeric variables are set to 0 and all the string variables are set to null. Sometimes a programmer wants to execute a **RUN** in one line of a program but to retain the value of one variable (perhaps a counter or a person's name). The value of a variable may be stored secretly in the video memory while this happens.

The program in Listing 3-S shows that the value of A$ is not retained when the RUN instruction is executed in line 150. Each time the program starts again, the value of A$ is null.

```
10 CLS
20 PRINT CHR$(23);
30 CLEAR 500
80 PRINT "NOW A$="; A$
90 INPUT "NEW VALUE FOR A$"; A$
100 IF A$ = "" THEN 90
150 RUN 30
```

Listing 3-S

Line	Description
10	Clears the screen.
20	Sets the screen in the 32 characters/line mode, rather than the regular 64 characters/line mode.
30	CLEARs 500 bytes of memory for string space.
80	PRINTs "NOW A$=" and the value of A$
90	Accepts the new value for A$ at the keyboard.
100	Returns to line 90 if A$ is null.
150	RUNs the program again starting at line 30 so the screen will not be cleared.

When the display is in the 32 characters/line mode, only the characters in the even-numbered video memory locations are displayed. When we POKE values into the odd-numbered locations in video memory, they are not displayed in this mode. And RUNning the program again will not affect their values unless the screen is cleared or otherwise changed. The lines in Listing 3-T show how this can be done for our small program.

```
40 GOTO 80
50 FOR J = 1 TO PEEK (15361)
60 A$ = A$ + CHR$ (PEEK (15361 + 2 * J ))
70 NEXT J
110 POKE 15361, LEN (A$)
120 FOR J = 1 TO LEN (A$)
130 POKE 15361 + 2 * J, ASC (MID$ (A$, J, 1))
140 NEXT J
150 RUN 50
```

Listing 3-T

(Note: Since lines 110 to 140 execute before lines 50 to 70, lines 110 to 140 will be described first.)

Line	Description
20	The first time the program is run, the program jumps around the routine in lines 50 to 70. (These lines read values from the screen, and the first time the program is run there are no values to be

read.)

110 After the string A$ is accepted from the keyboard, the length of the string is POKEd into the first **odd** position of the video memory. Since the screen is in the 32 characters/line mode, this and the POKEs to video memory in lines 120 to 140 are not displayed on the screen.

120 A loop is started with J as the looping variable. The loop goes to the number which represents the length of the string A$.

130 In the loop, the ASCII numeric representation of each character in the string A$ is POKEd into successive odd video memory locations. The value **15361** + 2 * J increases by 2 each time J is incremented by 1, so only the odd locations are used. Location 15361 contains the number that tells how many ASCII values are being POKEd into memory, and they are POKEd in to the odd locations that immediately follow 15361 in memory.

140 The loop ends.

50 Lines 50 to 70 execute each time the program is run, except the very first time. Line 50 begins a loop using J as the looping variable. It will go to PEEK(15361) which will contain the length of the string A$ that was POKEd in line 100.

60 The string A$ is reconstructed. Each value POKEd in by lines 120 to 140 is PEEKed out by this loop. Notice that the value **15361** + 2 * J is the same as on line 130.

70 The loop ends and the program proceeds to line 80.

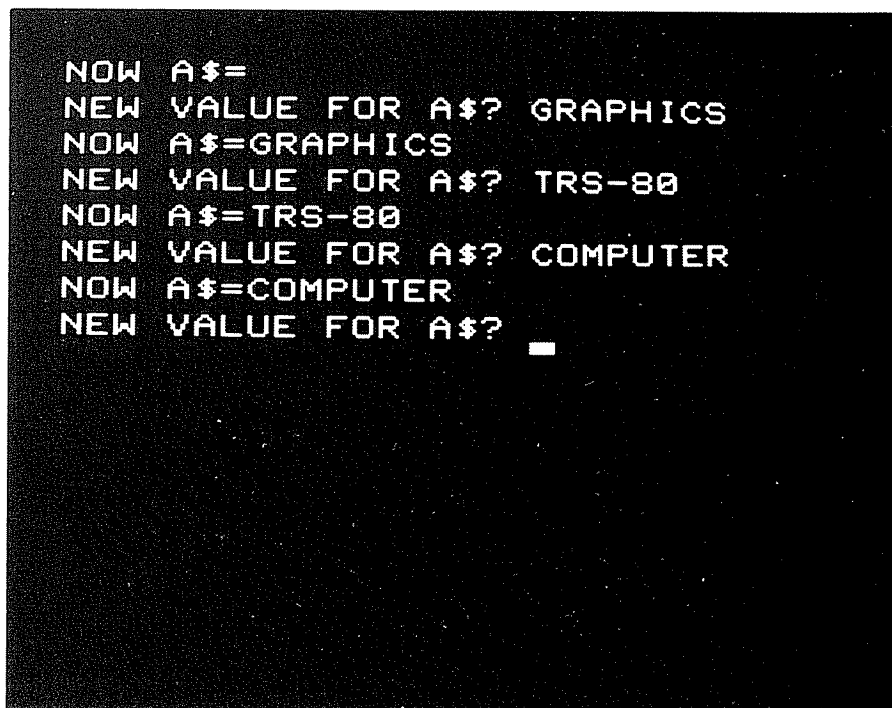


Photo 3-T

After you RUN this program, you may want to see the characters that are hidden by this routine. You can see them by following the following steps:

1) Type RUN <ENTER> to run the program.
2) Type in a word or a name and press <ENTER>.
3) When the program asks "NEW VALUE FOR A$?" again, press the <BREAK> key.
4) Type **PRINT CHR$(28)** and press <ENTER> to return the screen to the 64 characters/line mode. **At this point, the characters the program POKEd into the odd video memory locations are visible.**
5) You may type **PRINT CHR$(23)** and press <ENTER> again to hide the characters again.

Chapter Summary

The POKE instruction places a value into a memory location. Its format is **POKE n, m** where **n** is a memory location and **m** is a value between 0 and 255.

The memory locations 15360 to 16383 are the **video memory** (sometimes called "video RAM"). The values in these locations determine what is displayed on the screen of the TRS-80 microcomputer.

Horizontal lines can be drawn on the screen by POKEing graphic character values (between 128 and 191) to the video memory in adjacent memory locations. Vertical lines can be drawn by POKEing the values into locations at intervals of 64.

Complex pictures may be drawn with POKE instructions the same way they are with PRINT instructions. The values for the locations and the character values can be READ from DATA statements.

To improve control over keyboard input, a programmer may use an INKEY$ routine that POKEs characters into the video memory.

After a design has been drawn on the screen, the values can be PEEKed from the video memory to determine how to redraw the design.

In the 32 characters/line mode, characters corresponding to only the **even-numbered** video memory locations are displayed. Values POKEd into the **odd-numbered** locations of the video memory are retained while a CLEAR or RUN is executed.

Meeting the Chapter Objectives

Here are the chapter objectives for you review. Can you meet all the objectives?

A. Write the locations of the beginning and end of the video memory on the TRS-80 Model I, Model III, and Model 4.
B. Write an example (in proper format) of a POKE instruction which places a graphics character on the display.
C. Write a program that draws a vertical line on the display, using POKE graphics.
D. Write an example of a simple program with an INKEY$ instruction that POKEs values corresponding to keystrokes, to the video RAM.
E. Write a program that PEEKs each position of the video RAM and displays each value.
F. Write a program that hides a string of characters in the odd-numbered locations of the video memory while a RUN instruction is executed.

More Programming Practice

1. Using the POKE instruction, write a program that draws your initials on the screen.

2. Write a routine that uses the video memory and the INKEY$ instruction to accept keyboard input in the 32 characters/line mode.

3. Write a routine that allows the user to draw a design in the top left corner of the screen. The routine should read and display the locations and values of the characters used in the design.

4. Write a program that hides the value of a **number** in the video memory in the 32 characters/line mode. (Hint: Use the VAL and STR$ instructions to convert between numeric and string values.)

5. Write a program that prompts the user for a memory location and a value, then POKEs the value into the location. Without scrolling, it should repeat this until the BREAK key is pressed.

Chapter Checkup

1. In the program in Listing 3-G, how could the speed of the blink of the cursor character be slowed down to half as fast?

2. How does the line in Listing 3-I delete the right-most character from the string being built?

3. In the program in Listing 3-N, why does S$ have to be set to null in line 50?

4. The section of the drawing program in Listing 3-Q traps errors. What kind of error is most likely to occur while the program is running?

5. The program in Listings 3-P, 3-Q, and 3-R displays the values of the video memory on the video screen. How would you change the program to make it print out the values on a line printer? (Consider whether the program would still have to pause between each value.)

6. What memory location determines what is displayed on the third line down on the screen, the fourth position in from the left?

7. What are the two values that can be POKEd into a video memory location to erase any characters that appear on the screen at that location?

8. What happens when you try to POKE a negative value into a video memory location?

9. How many characters can be "hidden" in the video memory in the 32 characters/line mode?

10. "PRINT @" locations range from 0 to 1023, and corresponding video memory locations range from 15360 to 16383. If you type **PRINT @ 1024, "Dennis"** at your TRS-80 microcomputer, an error returns. But no error is returned if you type **POKE 16384, 65.** Why should the programmer be very careful about POKEing to locations beyond the video memory?

Chapter 4.
First Applications: Business and Education

OBJECTIVES:

At the end of this chapter, the reader will be able write a program with the following characteristics:

A. It PRINTs a string of graphic characters around the heading for an attractive title screen.
B. It calls an INKEY$ keyboard input routine for all keyboard input.
C. The program contains a main section that calls the other sections as subroutines. These subroutines should have only one entry (from the main section) and one exit (a RETURN). An exception is the keyboard input routine which is called as needed throughout the program.
D. The program READs numbers from DATA statements to build graphic shapes for a circle, a square, a rectangle, a triangle, and an oval.
E. The program displays one of the shapes (randomly chosen), asks the name of the shape, accepts an answer, and displays another shape. A total of twenty questions are asked.
F. The program keeps score and makes a bar graph to show how many of each shape the user identified correctly.

INTRODUCTION

Chapters 1, 2, and 3 discussed specific techniques of creating graphics on a TRS-80 Model I, III, or 4 screen. These methods are probably used in ninety percent or more of the TRS-80 BASIC programs written today. In this chapter you will find several applications programs that use these techniques. All the major components and many minor points of these programs are explained.

To gain the most from these sample programs, you should take the time to key them in at your TRS-80 keyboard. Look for the routines that have been discussed in the previous chapters and the variations of these routines. After you are comfortable with what each component of each program does, you may want to alter each program yourself. This is one of the best ways to prove to yourself that you really understand the techniques used.

THE STRUCTURE OF THE PROGRAMS

The programs presented in this chapter have several common features:

1) Each program has a main section which calls many subroutines. This structure makes if easy to organize, write, understand, and maintain the programs.

2) Each program begins with the keyboard input routine (in compressed form) from Chapter 3. Putting this routine before the main section of the program makes the keyboard input routine execute more quickly. This routine is called by the other subroutines whenever keyboard input is required.

3) Each of the subroutines is documented with the same label it is given when it is "called" from the main section of the program.

4) Each subroutine has only one entry and one exit. The main section of the program is the only part of the program that calls the subroutines, and the only exit is with a "RETURN" statement. (The only exception to this rule is the keyboard input routine, which is called as needed.)

Pay special attention to these features as you work with these sample programs.

PROGRAM 1: AN X,Y GRAPHING PROGRAM

The first program uses PRINT and POKE graphics to accept data at the keyboard and plot it on the screen. Lines 15 to 55 contain the main section of the program.

```
5     GOTO 15: 'X,Y Graphing Program by Dennis Tanner
10    '
       Lines 11 to 13 are the keyboard input routine from chapter 3

11    LO = PEEK(16416)+PEEK(16417)*256: L1=LO: I$=INKEY$: S$="":
      CU=32
12    CU=168-CU: POKE LO,CU: FOR JJ=1 TO 20: I$=INKEY$: IF I$=""
      THEN NEXTJJ: GOTO 12
      ELSE JJ=20: NEXTJJ: I=ASC(I$):IF I=8 AND S$<>""
      THEN POKE LO,32: LO=LO-1: S$=LEFT$(S$,LEN(S$)-1): GOTO 12
13    IF I=13 AND S$<>"" THEN POKE LO,32: PRINT: RETURN
      ELSE IF I<LV OR I>HV THEN 12
      ELSE IF LO-L1=LE THEN 12 ELSE POKE LO,I: S$=S$+I$: LO=LO+1:
      CU=32: GOTO12
14'
       Main section of the program

15    CLEAR 200: DIM NU(10,10): 'INITIALIZE VARIABLES
20    GOSUB 100 'PRINT HEADING
25    GOSUB 200 'ACCEPT PARAMETERS AT KEYBOARD
30    GOSUB 300 'ACCEPT THE DATA AT THE KEYBOARD
35    GOSUB 400 'CLEAR SCREEN AND PRINT X AXIS AND NUMBERS
40    GOSUB 500 'PRINT Y AXIS AND NUMBERS
45    GOSUB 600 'PLOT THE NUMBERS
50    GOSUB 700 'AWAIT KEY PRESS
55    GOTO 15
99 '
```

```
        Print Heading

100     CLS: PRINT@16,STRING$(32,140):
        PRINT @ 86, "X,Y Graphing  Program"
110     PRINT@148,"Written by Dennis Tanner":
         PRINT@208,STRING$(32,140)
120     RETURN
199 '
        Accept Parameters at Keyboard

200     PRINT@341,"X Interval? (1 - 10): " CHR$(30);: LE=2: LV=47: HV=58:
        GOSUB 11: XI=VAL(S$): IF XI<1 OR XI>10 THEN 200
210     PRINT@405,"Y Interval? (1 - 10): " CHR$(30);: LE=2: LV=47: HV=58:
        GOSUB 11: YI=VAL(S$): IF YI<1 OR YI>10 THEN 210
220     RETURN
299 '
        Accept Data at Keyboard

300     PRINT@972,"Type in numbers to graph, or 999 to end.";
310     FOR CO=1 TO 100
320             PRINT@538,"Number";CO
330             PRINT@602,"X: " CHR$(30);: LE=3: GOSUB 11: X1=VAL(S$):
                IF X1=999 THEN EN=CO-1: CO=100: NEXT CO: RETURN
                ELSE IF X1>10*XI THEN 330
340             PRINT@666,"Y: " CHR$(30);: GOSUB11: Y1=VAL(S$):
                IF  Y1=999 THEN EN=CO-1: CO=100: NEXT CO: RETURN
                ELSE IF Y1>10*YI THEN 340
350             PRINT@666,CHR$(30);
360             NU(INT(X1/XI),INT(Y1/YI)) = NU(INT(X1/XI),INT(Y1/YI))+1
370     NEXT CO
380     EN=100
390     RETURN
399'
        Clear Screen and Print X Axis and Numbers

400     CLS: FOR J=0TO10
410             PRINT@968+J*5,J*XI;
420     NEXT J
430     PRINT@904,STRING$(55,140);
440     RETURN
499'
        Print Y Axis and Numbers

500     FOR J=0TO10
510     PRINT@832-J*64,J*YI;
520     POKE 16198-J*64,191
530     NEXT J
540     RETURN
599 '
        Plot the Numbers
```

```
600  FOR J=0TO10
610        FOR K=0TO10
620         IF NU(J,K) <> 0 THEN PRINT@840+J*5-K*64, NU(J,K);
630        NEXT K
640  NEXT J
650  RETURN
699 '
      Await Key Press

700  PRINT@20,"Press any key to begin again.";: I$=INKEY$
710  I$=INKEY$: IF I$="" THEN 710
720  RETURN
```

Listing 4-A

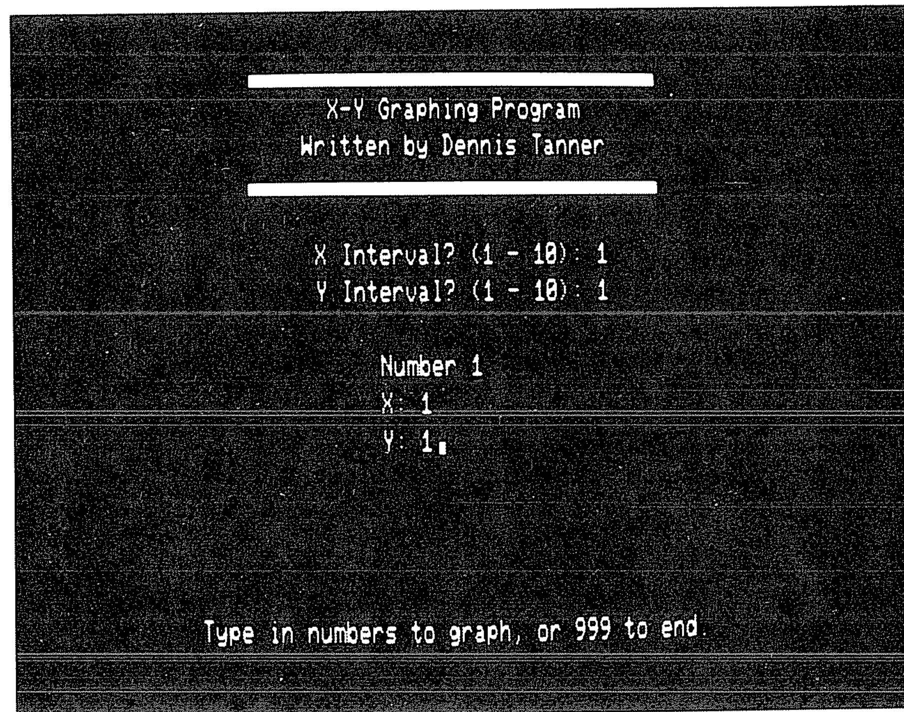


Photo 4-A-1

Here is a section-by-section description of the first program:

Line 5 causes the program to "jump around" the keyboard input subroutine.

Lines 11 to 13 contain the keyboard input subroutine discussed in Chapter 3.

Lines 15 to 55 contain the main section of the program. The overall logic of the program is shown in this section. Each of the subroutines is called in turn, and line 55 causes the program to begin again.

Notice that the initialization in line 15 is not a subroutine. The reason is, a CLEAR statement clears all the variables and the outstanding subroutine RETURN locations. So a CLEAR statement in a subroutine would cause a "Return without Gosub" error when the program tried to RETURN.

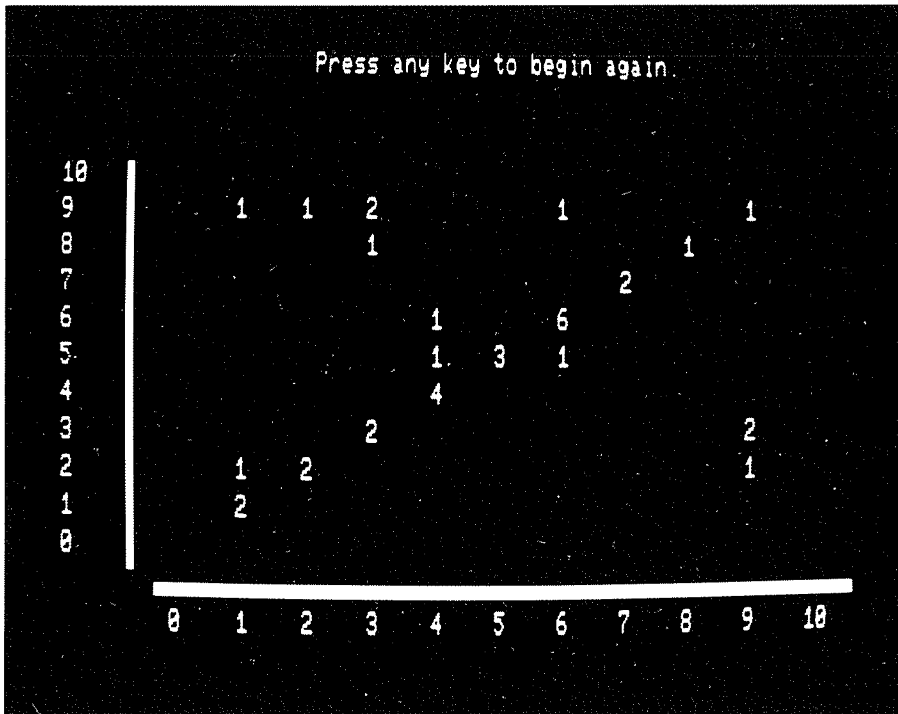


Photo 4-A-2

Lines 100 to 120 clear the screen and PRINT the heading. The STRING$ function is used in lines 100 and 110 to create the graphics around the heading.

Lines 200 to 220 PRINT questions and then accept responses using the keyboard input subroutine. The length used (LE) is 2. The low value (LV) and high value (HV) are 47 and 58 respectively, meaning that only numbers are accepted. The X interval value is assigned to XI, and the Y interval value is assigned to YI. If either value is less than 1 or greater than 10, it is rejected.

Lines 300 to 370 accept the data at the keyboard. This routine builds the array NU(x,y), where x is the X value of the pair, and y is the Y value. The array serves as a counter, so if there were 3 numbers in the position X=5 Y=7, then NU(5,7) would equal 3.

CO is the counter, so up to 100 X,Y pairs are accepted. EN is the number of pairs returned at the end of the routine. If the user types 999 for the item 4, there are CO - 1 or 3 of the X,Y pairs (lines 330 and 340). If the user puts in 100 pairs, EN is 100 (line 380).

If the X coordinate is greater than 10 times its interval (XI), or the Y coordinate is greater than 10 times its interval (YI), then the number is not accepted.

PRINTing CHR$(30) with a semicolon clears to the end of the current line. This character is PRINTed immediately preceding each call to the keyboard input routine. So if a value is not accepted for either X or Y, that value is cleared from the screen before another is accepted.

Since there are at most 10 intervals, the program keeps track of only 10 Y values per X value, and of only 10 X values. So if the X interval is 5 and the Y interval is 10, the values (20,20), (21,22), (23,20), (24,27), (23,26), and (22,28) would all be counted as (20,20), or (4x5, 2x10). Thus NU(4,2) would equal 6. Line 360 in the program puts each value in its proper location in the array.

Lines 400 to 440 PRINT the X axis and the numbers under it. In the J loop, 968+J*5 is the location (incremented by 5 each time). J*XI gives each value in the loop, incremented by the X interval.

Lines 500 to 540 POKE the Y axis into the video memory and PRINT the values beside it.

Lines 600 to 650 PRINT the values of the NU(x,y) array on the screen in the J loop. The horizontal position (in intervals of 5) is determined by adding J*5 to 840. The vertical position is determined by subtracting K*64 from the 840 + J*5.

Lines 700 to 720 PRINT a message at the bottom of the screen and wait for a keypress before RETURNing.

Because this program PRINTs the values of the array at the proper location, the screen presents a **visual numeric** display. The next program presents a more **graphic** display of a single-dimensioned array.

PROGRAM 2: A BAR GRAPH PROGRAM

This program shares many lines with the previous program. You may use the program in Listing 4-A, **except** for the lines marked with an

asterisk. Do not key in the asterisks as part of your program!

```
5*     GOTO 15: 'Bar Graph Program by Dennis Tanner
10     '
        Lines 11 to 13 are the keyboard input routine from Chapter 3

11     LO = PEEK(16416)+PEEK(16417)*256: L1=LO: I$=INKEY$: S$="":
       CU=32
12     CU=168-CU: POKE LO,CU: FOR JJ=1 TO 20: I$=INKEY$:
       IF I$="" THEN NEXTJJ: GOTO 12
       ELSE JJ=20: NEXTJJ: I=ASC(I$):IF I=8 AND S$<>""
       THEN POKE LO,32: LO=LO-1: S$=LEFT$(S$,LEN(S$)-1): GOTO 12
13     IF I=13 AND S$<>"" THEN POKE LO,32: PRINT: RETURN
       ELSE IF I<LV OR I>HV THEN 12
       ELSE IF LO-L1=LE THEN 12 ELSE POKE LO,I: S$=S$+I$: LO=LO+1:
        CU=32: GOTO12
14'
        Main section of the program

15*    CLEAR 200: DIM NU(10): 'INITIALIZE VARIABLES
20     GOSUB 100 'PRINT HEADING
25     GOSUB 200 'ACCEPT PARAMETERS AT KEYBOARD
30     GOSUB 300 'ACCEPT THE DATA AT THE KEYBOARD
35     GOSUB 400 'CLEAR SCREEN AND PRINT X AXIS AND NUMBERS
40     GOSUB 500 'PRINT Y AXIS AND NUMBERS
45     GOSUB 600 'PLOT THE NUMBERS
50     GOSUB 700 'AWAIT KEY PRESS
55     GOTO 15
99 '
        Print Heading

100*   CLS: PRINT@16,STRING$(32,140):PRINT @ 87,
       "Bar Graph Program"
110    PRINT@148,"Written by Dennis Tanner":
       PRINT@208,STRING$(32,140)
120    RETURN
199 '
        Accept Parameters at Keyboard

200    PRINT@341,"X Interval? (1 - 10): " CHR$(30);: LE=2: LV=47: HV=58:
       GOSUB 11: XI=VAL(S$): IF XI<1 OR XI>10 THEN 200
210    PRINT@405,"Y Interval? (1 - 10): " CHR$(30);: LE=2: LV=47: HV=58:
       GOSUB 11: YI=VAL(S$): IF YI<1 OR YI>10 THEN 210
220    RETURN
299 '
       Accept Data at Keyboard

300    PRINT@972,"Type in numbers to graph, or 999 to end.";
310    FOR CO=1 TO 100
320*           PRINT@538,"Number";
```

```
330*        PRINT@602,CO": "  CHR$(30);: LE=3: GOSUB 11:
            X1=VAL(S$):
            IF X1=999 THEN EN=CO-1: CO=100: NEXT CO: RETURN
            ELSE IF X1>10*XI THEN 330
360*        NU(INT(X1/XI)) = NU(INT(X1/XI))+1
370   NEXT CO
380   EN=100
390   RETURN
399'
       Clear Screen and Print X Axis and Numbers

400*  CLS: FOR J=1TO10
410*        PRINT@967+J*5,J*XI;
420   NEXT J
430   PRINT@904,STRING$(55,140);
440   RETURN
499 '
       Print Y Axis and Numbers

500*  FOR J=1TO10
510         PRINT@832-J*64,J*YI;
520         POKE 16198-J*64,191
530   NEXT J
540   RETURN
599 '
       Plot the Numbers

600*  FOR J=1TO10
610*        IF NU(J) > YI*10 THEN NU(J) = YI*10
620*        NY=INT(NU(J)/YI): IF NU(J)=0 THEN NEXTJ: RETURN
630*        FOR K=1 TO NY
640*         POKE 15360 + 840 + J*5 - K*64, 191
650*   NEXT K: NEXT J
660*  RETURN
699 '
       Await Key Press

700   PRINT@20,"Press any key to begin again.";: I$=INKEY$
710   I$=INKEY$: IF I$="" THEN 710
720   RETURN
```

Listing 4-B

As you can see, most of the lines in the program came from the program in Listing 4-A. Let's analyze the sections that contain differences.

Line 5 has the same function as before but a different label.

Lines 15 to 55 are almost identical to the program in Listing 4-A. The array NU(x) is dimensioned as a single-dimension array in this program

because we are specifying only the Y values, not both the X and Y values.

Lines 100 to 120 simply print different information in the heading.

Lines 300 to 370 are accepting single numbers, not pairs of numbers.

Lines 400 to 450 and Lines 500 to 540 plot the numbers from 1 to 10 instead of from 0 to 10.

Lines 600 to 650 contain significant differences from the previous program. In this program, for each of the 10 intervals, the variable NY is computed as the number to plotted as the frequency of Y. As seen in line 610, if the number is zero, no plotting occurs.

The K loop (lines 620 to 640) goes from 1 to the newly-computed Y freqency (NY). It POKEs that many graphic characters 191 into the proper screen position. Since the right number of characters is POKEd into position, each bar is the right height.

Three sections of the program have not changed, and the logic of the program has not changed at all. The modular format keeps the program easy to understand.

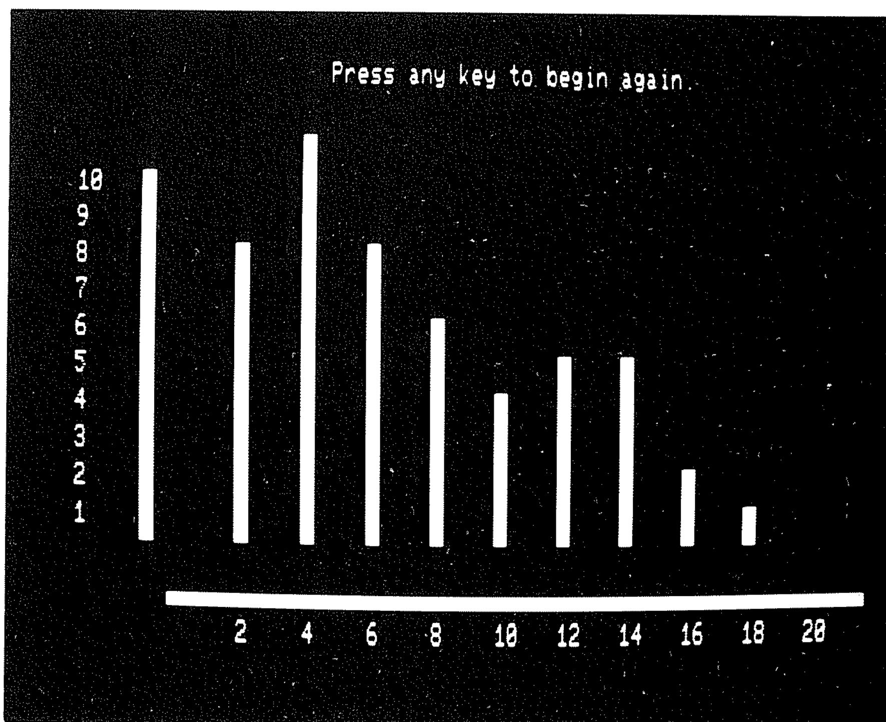


Photo 4-B

PROGRAM 3: A FANCIER BAR GRAPH PROGRAM

This program is an enhanced version of the program in Listing 4-B. The "Plot the Numbers" routine routine has changed to allow more accurate plotting of large numbers. It still contains the code to POKE in the values (lines 600 to 630 of Listing 4-C). But it also contains code to use the SET instruction to produce more accurate graphs when the interval is greater than 1.

For this program, just delete lines 600 to 660 of Listing 4-B. Then insert these lines:

```
600  IF YI>1 THEN 635 ELSE FOR J=1 TO 10
605   IF NU(J)> YI*10 THEN NU(J)= YI*10
610   NY= INT(NU(J)/YI): IF NU(J)=0 THEN NEXTJ: RETURN
615   FOR K = 1 TO NY
620    POKE 15360 + 840 + J*5 - K*64, 191
625   NEXT K: NEXT J
630  RETURN
635  FOR J=1 TO 10: IF NU(J)=0 THEN NEXTJ:RETURN
640   IF NU(J) > YI*10 THEN NU(J)= YI*10
645   NY = INT((NU(J)/YI)*3)
650   FOR K = 1 TO NY
655    SET (17 + 10*J, 40-K): SET (18 + 10*J, 40-K)
660   NEXT K: NEXT J
665  RETURN
```

Listing 4-C

This time we use the SET instruction in line 655 to create the graphics. The Y height is computed in line 645. The Y counter **NU(J)** is divided by the interval to give the number of blocks to plot. Then it is divided by 3 since each character-sized block has 3 vertical pixel positions. This number NY is used as the counter in the K loop that SETs the graphics points.

The first three programs have shown possible business applications of TRS-80 graphics. The remaining two programs demonstrate educational uses of graphics.

PROGRAM 4: GRAPH-READING PRACTICE

This program uses the same basic structure as the last two programs, but it generates its own data and asks the user about the graph.

```
5     GOTO 15: 'Graph Reading Program by Dennis Tanner
10    '
       Lines 11 to 13 are the keyboard input routine from Chapter 3

11    LO = PEEK(16416)+PEEK(16417)*256: L1=LO: I$=INKEY$: S$="":
```

```
       CU=32
12     CU=168-CU: POKE LO,CU: FOR JJ=1 TO 20: I$=INKEY$:
       IF I$="" THEN NEXTJJ: GOTO 12
       ELSE JJ=20: NEXTJJ: I=ASC(I$):
       IF I=8 AND S$<>"" THEN POKE LO,32: LO=LO-1:
       S$=LEFT$(S$,LEN(S$)-1): GOTO 12
13     IF I=13 AND S$<>"" THEN POKE LO,32: PRINT: RETURN
       ELSE IF I<LV OR I>HV THEN 12
       ELSE IF LO-L1=LE THEN 12 ELSE
       POKE LO,I: S$=S$+I$: LO=LO+1: CU=32: GOTO12
14'
        Main section of the program

15     CLEAR 200: DIM NU(10): 'INITIALIZE VARIABLES
20     GOSUB 100 'PRINT HEADING
25     GOSUB 200 'SET PARAMETERS FOR PROGRAM
30     GOSUB 300 'GENERATE RANDOM DATA
35     GOSUB 400 'CLEAR SCREEN AND PRINT X AXIS AND NUMBERS
40     GOSUB 500 'PRINT Y AXIS AND NUMBERS
45     GOSUB 600 'PLOT THE NUMBERS
50     FOR QU = 1 TO 5
55            GOSUB 700 'GENERATE AND DISPLAY QUESTION
60            GOSUB 800 'ACCEPT RESPONSE AT KEYBOARD
65            IF RE <> AN THEN GOSUB 900: GOTO 60
                   'IF INCORRECT, THEN DISPLAY ERROR MESSAGE
                    AND ACCEPT ANOTHER RESPONSE
70            GOSUB 1000 'POSITIVE REINFORCEMENT MESSAGE
75     NEXT QU
80     GOSUB 1100 'DISPLAY REPORT AND AWAIT KEYPRESS
85     GOTO 15
99 '
       Print Heading

100    CLS: PRINT @ 16, STRING$(32,140): PRINT @ 85,
       "Graph Testing Program"
110    PRINT @ 148, "Written by Dennis Tanner":
       PRINT @ 208, STRING$(32,140)
120    PRINT @ 979, "Press <ENTER> to continue";
130    I$=INKEY$
140    I$=INKEY$: IF I$ <> CHR$(13) THEN 140
150    RETURN
199 '
       Set Parameters for Program

200    X1 = 10: Y1 = 10
210    XI = 1: YI = 1
220    RETURN
299 '
       Generate Random Data

300    FOR J = 1 TO 10
```

```
310         NU(J) = RND (10)
320     NEXT J
330     RETURN
399 '
        Clear Screen and Print X Axis and Numbers

400     CLS: FOR J = 1 TO 10
410         PRINT @ 967 + J * 5, J * XI;
420     NEXT J
430     PRINT @ 904, STRING$(55,140)
440     PRINT @ 960, "Day > > >";
450     RETURN
499 '
        Print Y Axis and Numbers

500     FOR J = 1 TO 10
510         PRINT @ 832 - J * 64, J * YI;
520         POKE 16198 - J * 64, 191
530     NEXT J
540     PRINT @ 128, "Candy Bars";
550     RETURN
599 '
        Plot the Numbers

600     FOR J = 1 TO 10
610         NY = INT(NU(J)/YI): IF NU(J)=0
            THEN NEXT J: RETURN
620         FOR K = 1 TO NY
630          POKE 15360 + 840 + J * 5 - K * 64, 191
640         NEXT K: NEXT J
650     RETURN
699 '
        Generate and Display Question

700     DA = RND (10): AN = NU(DA): R1 = 1
710     PRINT @ 10, "How many candy bars were eaten on day" DA "? ";
        CHR$(30);
720     RETURN
799 '
        Accept Response at Keyboard

800     LE = 2: LV = 48: HV = 57: GOSUB 11
810     RE = VAL(S$)
820     RETURN
899 '
        Display Error Message
900     IF RE < AN THEN
            PRINT @ 85, "Sorry," RE "is too low.";: GOTO 920
910     PRINT @ 85, "Sorry," RE "is too high.";
920     PRINT @ 50 + LEN(STR$(DA)), CHR$(30);
930     R1 = 0
```

```
940   RETURN
999 '
      Positive Reinforcement Message

1000  PRINT @ 64, CHR$(30);
1010  PRINT @ 86, "Yes," AN "is correct.";
1020  FOR K = 1 TO 500: NEXT K
1030   PRINT @ 64, CHR$(30);
1040  IF R1 = 1 THEN
             NR = NR + 1
1050 RETURN
1099 '
      Display Report and Await Keypress

1100  PRINT @ 0, CHR$(30); PRINT @ 64, CHR$(30) ;
1110  PRINT @ 3, "On this graph, you got" NR
             "out of 5 right on the first try.";
1120  PRINT @ 80, "Press <ENTER> for another graph.";
1130 I$=INKEY$
1140 I$=INKEY$: IF I$ <> CHR$(13) THEN 1140
1150 RETURN
```

Listing 4-D

Here is a section-by-section description of this educational program:

Line 5 "jumps" around the keyboard input routine to the main section of the program.

Lines 11 to 14 are the keyboard input routine from Chapter 3.

Lines 15 to 85 contain the main section of the program. All the subroutines below are called from this main section. Lines 50 to 75 hold a FOR-NEXT loop for the questions asked by the program. Line 65 checks the correctness of the response and branches back to line 60 until the response is right. This branching within the main section eliminates the need for jumping directly from subroutine to subroutine in the sections below.

Lines 100 to 160 print the heading much as before.

Lines 200 to 220 set the parameters for the graph. In the previous two programs, these values were accepted as data at the keyboard, but this program sets them the same for every graph drawn.

Lines 300 to 330 generates random data for the graph. This data was accepted at the keyboard in the previous programs, but here it is randomly generated to test the user's understanding of graphs.

Lines 400 to 450 clear the screen and print the X axis as before. This time the X axis is labelled as "Day > > >" for the test.

Lines 500 to 550 print the Y axis and numbers with the additional "Candy Bars" label.

Lines 600 to 650 plot the values as a bar graph exactly as before.

Lines 700 to 720 choose a day (DA) and ask how many candy bars were eaten on that day. The variable R1 (which has a value of 1 if the user gets the problem correct on the first try and a value of 0 if not) is initialized for each problem with a value of 1.

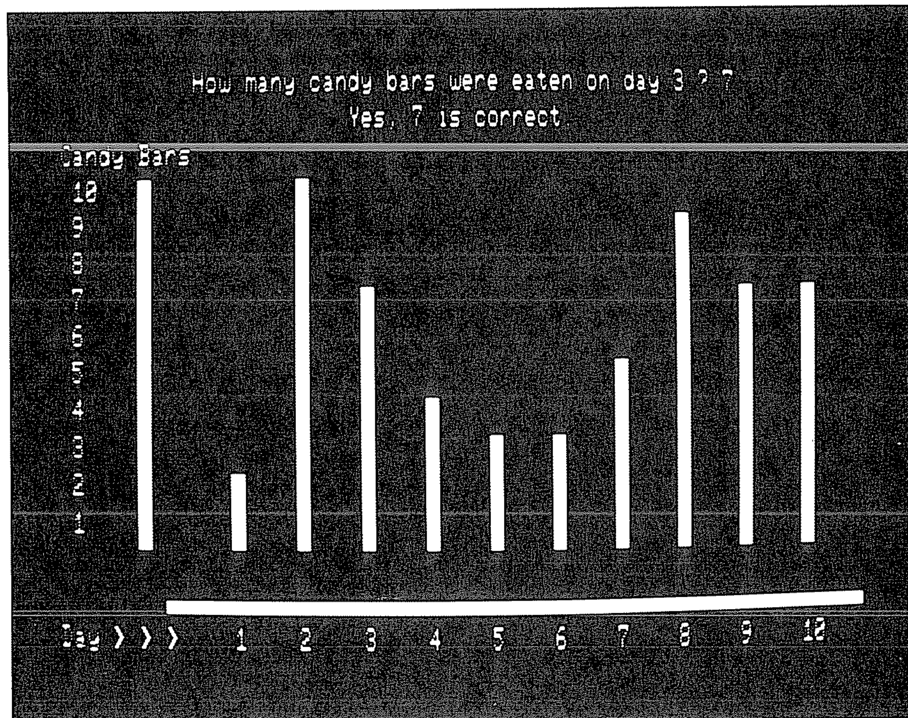


Photo 4-D

Lines 800 to 820 call the keyboard input routine and store the user's response in variable RE.

Lines 900 to 940 print appropriate messages for user responses less than or more than the correct answer. The variable R1 is set at 0 since the user did not get the answer right on the first try.

Lines 1000 to 1050 print a reinforcement message and then pause. The variable NR (number right) is incremented if the user got the problem right on the first try (if R1=1).

Lines 1100 to 1150 display a report of the number correct, and

await a keypress.

PROGRAM 4 is another example of changing the program slightly to perform another task.

PROGRAM 5: ADDITION DRILL WITH LARGE CHARACTERS

The final sample program in this chapter uses large characters stored in a string array to present addition problems to students.

```
5     GOTO 20 ' Large number addition drill
10    '
       Lines 11 to 13 are the keyboard input routine from Chapter 3

11    LO = PEEK(16416)+PEEK(16417)*256: L1=LO: I$=INKEY$: S$="":
      CU=32
12    CU=168-CU: POKE LO,CU: FOR JJ=1 TO 20: I$=INKEY$:
      IF I$="" THEN NEXTJJ: GOTO 12
      ELSE JJ=20: NEXTJJ: I=ASC(I$):
      F I=8 AND S$<>"" THEN POKE LO,32: LO=LO-1:
      S$=LEFT$(S$,LEN(S$)-1): GOTO 12
13    IF I=13 AND S$<>"" THEN POKE LO,32: PRINT: RETURN
      ELSE IF I<LV OR I>HV THEN 12
      ELSE IF LO-L1=LE THEN 12
      ELSE POKE LO,I: S$=S$+I$: LO=LO+1: CU=32: GOTO12
14 '
      Main body of the program
20    CLEAR 1000: GOSUB 100 'INITIALIZE VARIABLES AND BUILD
                            CHARACTERS
25    GOSUB 200 'PRINT HEADING
30    GOSUB 300 'ACCEPT NUMBER OF PROBLEMS AND MAXIMUM
                            ADDEND SIZE
35    FOR PR = 1 TO NP 'LOOP FOR PROBLEMS
40          GOSUB 400 'GENERATE PROBLEM
45          GOSUB 500 'DISPLAY PROBLEM
50          GOSUB 600 'ACCEPT ANSWER
55          IF RE <> AN THEN
            GOSUB 700: GOTO 50 'IF THE ANSWER ISN'T RIGHT,
                        PRINT MESSAGE AND TRY AGAIN
60          GOSUB 800 'PRINT REINFORCEMENT MESSAGE
                        FOR CORRECT ANSWER
65    NEXT PR 'END OF LOOP FOR PROBLEMS
70    GOSUB 900 'DISPLAY REPORT
75    GOSUB 1000 'AWAIT KEYPRESS
80    END
99 '
      Initialize Variables and Build Characters

100   DEFINT A-Z: DIM NU$(10)
105   CLS: PRINT @ 980, "< < < Please wait > > >";
```

```
110   FOR NU = 0 TO 10
115          FOR RO = 1 TO 3
120                 FOR PO = 1 TO 5
125                        READ A: NU$(NU) = NU$(NU) + CHR$(A)
130                 NEXT PO
135                 NU$(NU) = NU$(NU) + CHR$(26) + STRING$(5,8)
140          NEXT RO
145   NEXT NU
150   RETURN
155 '
      This data (read in line 125) is used to create
      the large numbers and the plus sign
160   DATA 184,159,143,175,180,
            191, 32, 32, 32,191,
            139,189,188,190,135
161   DATA  32,181,191, 32, 32,
             32, 32,191, 32, 32,
             32,188,191,188, 32
162   DATA 160,190,143,191,148,
             32,160,190,135, 32,
            168,191,189,188,148
163   DATA 168,159,143,191,180,
             32,136,174,191,145,
            139,189,188,191,135
164   DATA 191,149, 32,191,149,
            191,189,188,191,189,
             32, 32, 32,191,149
165   DATA 191,159,143,143,143,
            143,143,143,191,180,
            139,189,188,191,135
166   DATA 184,159,143,143,132,
            191,159,143,175,180,
            139,189,188,190,135
167   DATA 138,143,143,175,191,
             32, 32, 32,190,151,
             32, 32,170,191, 32
168   DATA 190,151,131,171,189,
            190,159,143,175,189,
            139,189,188,190,135
169   DATA 184,159,143,175,180,
            139,189,188,190,191,
            172,180,184,190,135
170
' The next line has data for the "+" sign.

171   DATA  32, 32,188, 32, 32,
            188,188,191,188,188,
             32, 32,191, 32, 32
199 '
      Print Heading
```

```
200   CLS: PRINT @ 20, STRING$(24,140)
210   PRINT @ 152, "Addition Problems"
220   PRINT @ 276, STRING$(24,140)
230   RETURN
299 '
      Accept Number of Problems and
             Maximum Addend Size

300   PRINT @ 464, "How many problems? (1 - 10): "; CHR$(30);: LV=48:
      HV=57: LE=2: GOSUB 10
310   NP = VAL(S$): IF NP > 10 OR NP = 0 THEN 300
320   PRINT @ 593, "Maximum addend (1 - 999): "; CHR$(30);: LE=3:
      GOSUB 10
330   MX = VAL(S$): IF MX = 0 THEN 320
340   RETURN
399 '
      Generate Problem

400   A1 = RND (MX): A2 = RND (MX)
410   RETURN
499 '
      Display Problem

500   A$(1) = RIGHT$(STR$(A1),LEN(STR$(A1))-1)
510   A$(2) = RIGHT$(STR$(A2),LEN(STR$(A2))-1)
520   R1 = 1: CLS: FOR J = 1 TO 2
530          FOR K = 1 TO LEN(A$(J))
540                 PRINT @ 102 - 10*(LEN(A$(J))) + 10*K +
                    256*(J-1), NU$(VAL(MID$(A$(J),K,1)));
550          NEXT K
560   NEXT J
570   PRINT @ 358 - 10*LEN(A$(2)), NU$(10);
580   PRINT @ 612 - 10*LEN(A$(2)),
      STRING$(10*(LEN(A$(2))+1),140);
590   RETURN
599 '
      Accept Answer

600   PRINT @ 770, "Your answer: ";CHR$(31);: LE = 4: GOSUB 10
610   RE = VAL(S$): AN = A1 + A2
620   RETURN
699 '
      If answer is not right, use this routine

700   R1 = 0: IF RE < AN THEN PRINT @ 896,
      "Sorry. Your answer is too low.";: GOTO 720
710   PRINT @ 896, "Sorry. Your answer is too high.";
720   PRINT @ 960, "Press <ENTER> to try again.";
730   I$=INKEY$:
740   I$=INKEY$: IF I$<>CHR$(13) THEN 740
750   RETURN
```

```
799 '
      Print Reinforcement Message for Correct Answer

800   PRINT @ 768, CHR$(31);
810   AN$ = RIGHT$(STR$(AN),LEN(STR$(AN))-1)
820   FOR K = 1 TO LEN(AN$)
830          PRINT @ 742 - 10*LEN(AN) + 10*K,
             NU$(VAL(MID$(AN$,K,1)));
840   NEXT K
850   IF R1 = 1 THEN NR = NR + 1
860   PRINT @ 976, "Right. Press <ENTER> to continue.";
870   I$=INKEY$
880   I$=INKEY$: IF I$<>CHR$(13) THEN 880
890   RETURN
899 '
      Display report

900   CLS: PRINT @ 23, "Addition Problems";:
      PRINT @ 197, "Problems tried:";
905   NP$ = RIGHT$(STR$(NP),LEN(STR$(NP))-1)
910   FOR K = 1 TO LEN(NP$)
915          PRINT @ 180 - 10*LEN(NP$) + 10*K,
             NU$(VAL(MID$(NP$,K,1)));
920   NEXT K
925   PRINT @ 453, "Problems correct:";
930   NR$ = RIGHT$(STR$(NR),LEN(STR$(NR))-1)
935   FOR K = 1 TO LEN(NR$)
940          PRINT @ 436 - 10*LEN(N4$) + 10*K,
             NU$(VAL(MID$(NR$,K,1)));
945   NEXT K
950   PC = INT ((NR*100/NP)+.5)
955   PC$ = RIGHT$(STR$(PC),LEN(STR$(PC))-1)
960   PRINT @ 709, "Percent correct:";
965   FOR K = 1 TO LEN(PC$)
970          PRINT @ 962 - 10*LEN(PC$) + 10 * K,
             NU$(VAL(MID$(PC$,K,1)));
975   NEXT K
980   PRINT @ 977, "Press <ENTER> to end program.";
985   RETURN
999 '
      Await Keypress
1000  I$=INKEY$
1010  I$=INKEY$
1020  CLS
1030  RETURN
```

Listing 4-E

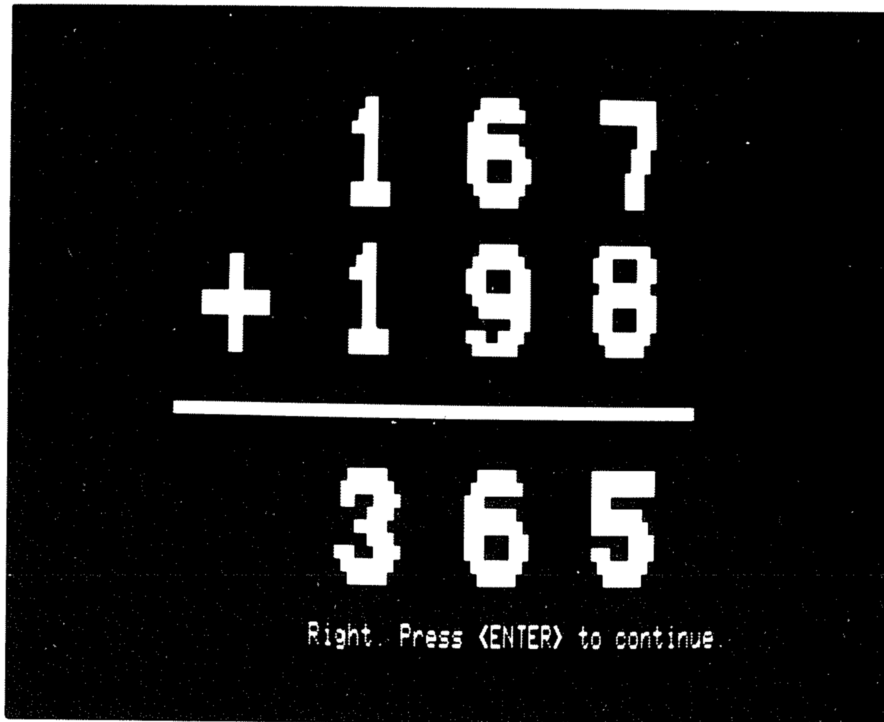


Photo 4-E

Here is a section-by-section description of the addition program:

Line 5 skips around the keyboard input routine.

Lines 11 to 13 are the same keyboard input routine we used before, at the beginning of the program to assure rapid execution.

Lines 20 to 80 contain the main section of the program. Notice the FOR-NEXT loop in lines 35 to 65 which contains all the branching for correct and incorrect responses.

Lines 100 to 150 initialize the variables and build the graphics blocks which make up the large character set. First the "Please wait" message appears so the user will know the program is executing even though nothing happens on the screen. The three nested loops build the number (the NU loop), the row (the RO loop), and the position with in the row (the PO loop). The PO loop reads successive numbers from the DATA statements in lines 160 to 171 and assigns characters with those values to the proper string variable in the NU$() array. Line 160 has the data for the "0" character, line 161 for the "1" character, and so on. The data for the plus sign is in line 171. (These large characters were created using the program in Listings 3-P, 3-Q, and 3-R. The author drew the

characters with the arrow keys and let the computer do the work of figuring which characters were needed. Like the ad says, computers should work and people should think!)

Lines 200 to 230 print the heading on the first screen.

Lines 300 to 340 use the keyboard input routine in lines 11 to 13 to accept the number of problems and the maximum size of the addend (the numbers to be added).

Lines 400 and 410 generate the problem using the maximum number (accepted in the previous subroutine) as the upper limit.

Lines 500 to 590 display the problem one character at at time on the screen. Let's analyze this section more thoroughly.

Lines 500 and 510 create a string version of the numeric variables generated as the addends of the problems. The STR$ function adds a leading space to the numeral, so the RIGHT$ function is used to "peel" the leading space off the string.

Line 520 initializes R1 (the "right the first time" variable) to 1. It then clears the screen and begins the loop which will print the two addends on the screen (the J loop).

Line 530 starts the loop for printing the individual addends (the K loop). The loop goes from 1 to the length of the string which contains the addend.

Line 540 PRINTs the graphics blocks that form the numerals. The position for each character is determined by a formula which includes the length of the addend string (LEN(A$(J))), the position within the string (K) and which addend it is (J). The block printed is derived from the specific character within the addend string (MID$(A$(J),K,1)), the value of that character, and finally the block within the NU$() array that corresponds with that value.

Lines 550 and 560 close up the K and J loops.

Line 570 prints the plus sign (NU$(10)).

Line 580 prints the line drawn under the problem.

Line 590 returns from the subroutine to the main section of the program.

Lines 600 to 620 accept the user's answer and assign it to the variable RE. Line 610 also computes the correct answer and assigns it to AN.

Lines 700 to 750 set R1 (the "right the first time" variable) to 0,

since this routine is only accessed if the user misses the answer. An appropriate error message is displayed, and the routine waits for the user to press the ENTER key (returned as character 13).

Lines 800 to 890 PRINT the correct answer (in big characters) and a reinforcement response when the user gets the answer correct. The routine used to PRINT the large characters is very similar to the one is lines 500 to 560. Line 850 increments the NR (number right) variable if R1 (the "right the first time" variable) is still 1. Lines 860 through 880 PRINT a message and await a press of the ENTER key.

Lines 900 to 985 display the report of the number worked, the number correct, and the percent correct. Each of these values is printed using the large block letters. The percent is rounded up if the decimal is .5 or greater, and rounded down if the decimal is less than .5 in line 950.

Lines 1000 to 1030 await a keypress before program ends.

Chapter Summary

Chapters 1, 2, and 3 discussed methods of creating graphics on the TRS-80 screen using SET/RESET/POINT graphics, using PRINT graphics, and POKEing to the video memory. The five programs in this chapter present useful applications of these graphics methods. To learn more from these applications programs, key them into a TRS-80 Model I, III, or 4 and try your own variations of the techniques shown.

Meeting the Chapter Objectives

For this chapter, the objectives are stated in the form of specifications of a program for the user to write. Can you now write a program that meets these specifications?

A. It PRINTs a string of graphic characters around the heading for a more attractive title screen.
B. It calls an INKEY$ keyboard input routine for all keyboard input.
C. The program contains a main section that calls the other sections as subroutines. These subroutines should have only one entry (from the main section) and one exit (a RETURN). An exception is the keyboard input routine which is called as needed throughout the program.
D. The program READs numbers from DATA statements to build graphic shapes for a circle, a square, a rectangle, a triangle, and an oval.
E. The program displays one of the shapes (randomly chosen), asks the name of the shape, accepts an answer, and displays another shape. A total of twenty questions are asked.
F. The program keeps score and makes a bar graph to show how many of each shape the user identified correctly.

More Programming Practice

1. Write a program that asks the user to type a name, then prints the name on the screen with large block letters.
2. Write an keyboard input routine that accepts consecutive characters in a vertical column instead of the usual horizontal row.
3. Write a program that gives random math problems with addends from 1 to 9, displays the numerals in large characters, and displays graphics blocks for counting beside the numbers. For example, numeral 2 would have two graphics blocks beside it.
4. Write a bar graph program that displays horizontal bars instead of vertical bars.
5. Write a program that generates a random number, displays the number, displays that random number of small graphic triangles, then begins again with another random.number.

Chapter Checkup

1. How does the modular programming format make programs easier to maintain?

2. In line 330 of Listing 4-A, the keyboard input routine is called (GOSUB 11) without first setting the high value and low value for acceptable keystrokes. Why was this not necessary in this line?

3. In the "Clear Screen and Print X Axis and Numbers" routine (lines 400-440) in Listing 4-A, which line of the program actually draws the graphic line for the X axis?

4. In the program in Listing 4-A, what number is printed if the value of NU(J,K) is 0 in line 620?

5. In the program in Listing 4-C, what is the purpose of line 640?

6. In Listing 4-C, why are there two SET instructions in line 655?

7. In the program in Listing 4-D, why are the "greater than" symbols used after "Day" in line 440?

8. What purpose does line 920 in Listing 4-D serve?

9. Lines 110 to 145 build the graphics blocks for the large numbers used in Listing 4-E. What part does line 135 play in this building process?

10. Line 330 of Listing 4-E checks to be sure the value returned for the maximum addend is not zero. Why is there no need to check that the value is not greater than 999?

Chapter 5
Printer Graphics

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write a program that prints a simple picture on a printer, using a series of LPRINT instructions. The data for this picture will be literal information stored in text strings.

B. Write a program that draws a sine wave on a printer. The TAB function used with mathematical calculations will be used to determine the print locations.

C. Write a program that draws a graph on the printer. The data determining the LPRINT locations in this program will be stored in a numeric array.

D. Write a program that prints one large graphic letter or number on a printer. String arrays will store the data for this program.

E. Write a subroutine that prints all the text characters (but not the graphics characters) from the TRS-80 screen to the keyboard. The text characters on the screen will determine the graphics in this program.

F. Write a program that reads the graphics from a portion of the screen and prints them on a printer. The graphics characters on the screen will determine what's printed in this program.

INTRODUCTION

The video display screen is the output device usually used for the display of graphics. But another useful device for displaying graphics is the printer.

The BASIC instructions for displaying text characters on a printer are almost identical to those used for displaying text characters on the video screen. The major difference is that an **LPRINT** instruction is used instead of **PRINT**. So the program in Listing 2-D which prints a tree with leaves and roots and a trunk works fine on a printer if the **PRINTs** are changed to **LPRINTs** as follows:

```
10   CLS
20   LPRINT"          LEAVES"
30   LPRINT"        LEAVESLEAVES"
40   LPRINT"    LEAVESLEAVESLEAVESLEAVES"
50   LPRINT"   LEAVESLEAVES  BIRD  LEAVES"
60   LPRINT"  LEAVESLEAVESLEAVESLEAVESLEAVES"
70   LPRINT"   LEAVESLEAVESLEAVESLEAVESLEA"
80   LPRINT"    LEAVESLEAVESLEAVESLEAVES"
90   LPRINT"        LEAVESLEAVES"
100  LPRINT"          TRUNK"
110  LPRINT"          TRUNK"
120  LPRINT"          TRUNK"
130  LPRINT"          TRUNK"
```

```
140   LPRINT"          ROOTSROOTS
150   GOTO 150
```

Listing 5-A

Line 10 and line 150 could be omitted from this program without affecting the graphics printout.

Programs (like the program in Listing 2-C) which print text graphics on the screen using the **PRINT @** instruction must be modified further for output to the printer. Since **LPRINT @** is not supported by the TRS-80 print functions, the programmer must use **LPRINT TAB** or a similar instructions when transferring these programs to the printer.

The program above stored the data determining what was LPRINTed in literal text strings (strings that are to be LPRINTed just as they appear). In this chapter we will also use mathematical calculations, numeric arrays, string arrays, the text on the screen, and the graphics on the screen to determine what will be LPRINTed.

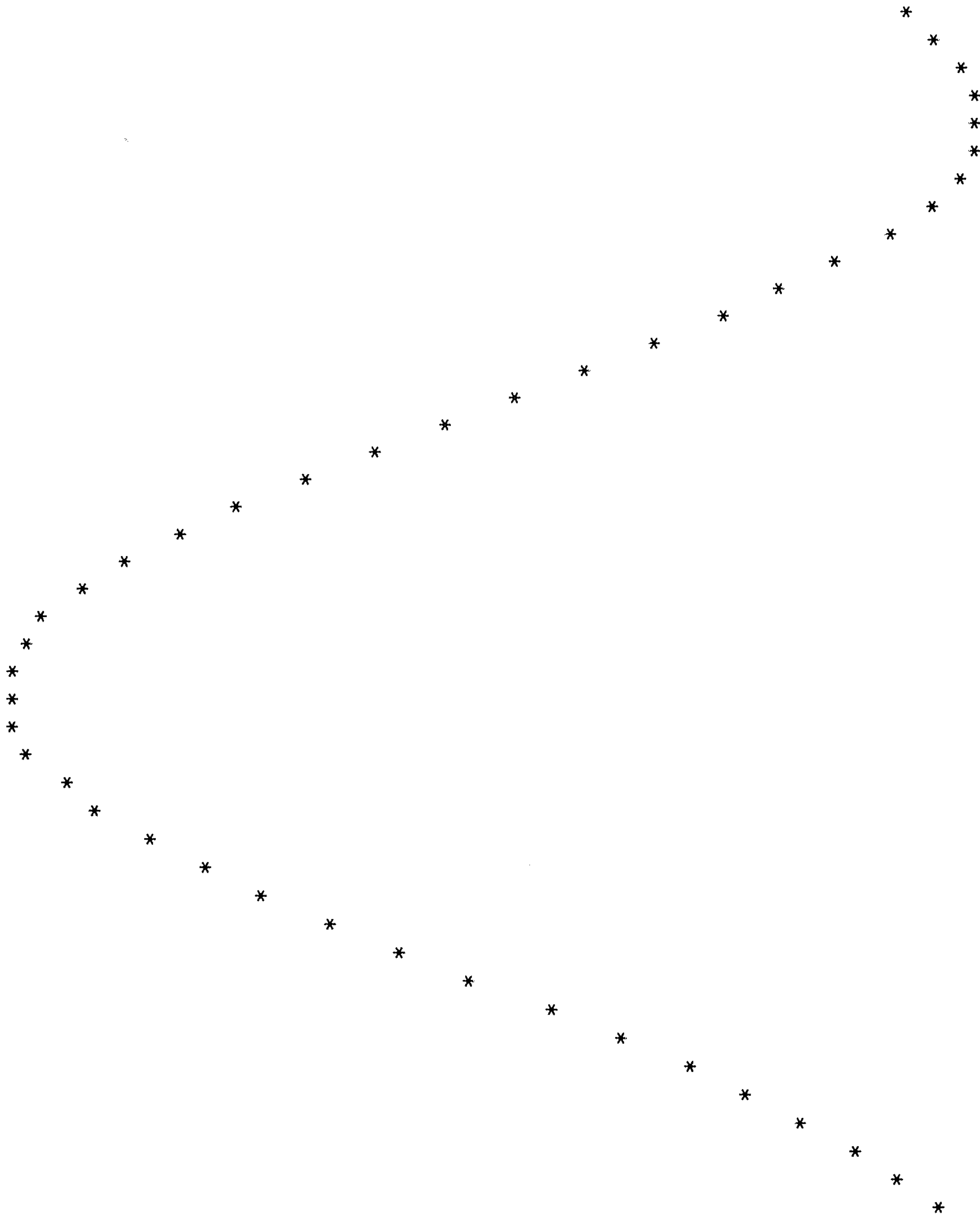
The programs and techniques mentioned in this chapter work on all standard 80- or 132-column line printers unless otherwise noted. The word "printer" is used throughout this chapter to refer to these standard printers.

CONTROLLING THE PRINTER: A SINE WAVE PROGRAM

On the printer, as on the video display, the position of the character printed is as important as the character itself. The program in Listing 5-B shows an example of the use of mathematical calculations with the TAB instruction in controlling the LPRINT position.

```
10    'Sine wave printer program
20    CLS
30    PRINT @ 76, "Press <ENTER> when the printer is
      ready."
40    I$ = INKEY$
50    I$ = INKEY$: IF I$ <> CHR$(13) THEN 50
60    CLS
70    PRINT @ 211, "Now printing sine wave..."
80    FOR J=1 TO 7.5 STEP .15
90        LPRINT TAB(40 + 35 * SIN(J)); "*"
100       PRINT 40 + 35 * SIN(J);
110   NEXT J
```

Listing 5-B



Printout 5-B

Here is a line-by-line analysis of the program:

Line	Description
10	Remarks.
20	Clears the screen.
30	PRINTs message on video display.
40	Catches any prior keystrokes with INKEY.
50	Accepts a keystroke (INKEY$) and checks to see if it is character 13 (the ENTER key). If not, the line is repeated until ENTER is pressed.

(Lines 30 to 50 make sure the printer is ready before the printing begins.)

60	Clears the screen again.
70	PRINTs a message on the video display telling that the LPRINTing on the printer is beginning.
80	Begins a FOR-NEXT loop. Since BASIC's sine function works with radians (rather than degrees), 2 x PI is the interval needed for one complete cycle. 2 x PI is about 6.3. This loop uses 7.5 to include one cycle plus a little more.
90	Since the sine function cycles between -1 and 1, the calculation for the amount of the TAB starts at 40 (the center point of an 80-column printer). The sine of the number is multiplied by 35 and that product is added to 40. The smallest result would 40+35*(-1) or 5, and the largest would be 40+35*1, or 75. This keeps the TAB number within the printer's range of 1 to 80. An asterisk is LPRINTed at the proper TAB position.
100	The value calculated in line 90 is displayed for the user to examine as the asterisk is LPRINTed.
110	The FOR-NEXT loop is ended.

In the program above, a mathematical function is used to calculate the position of the character. The program that follows uses data entered at the keyboard and stored in a numeric array to determine the positions of the characters.

A BAR GRAPH PROGRAM FOR THE PRINTER

The program in Listing 5-C accepts data at the keyboard and graphs it, as do some of our previous programs. But this time, the graph is LPRINTed on the printer instead of the video display. In this program, note that the data are stored in a numeric array and "peeled off" as the printout progresses.

```
5     GOTO 15:' Printing Bar Graph Program by D. Tanner
10    '
```

Lines 11 to 13 are the INKEY$ routine from Chapter 3.

```
11    LO=PEEK(16416)+PEEK(16417)*256:L1=LO:I$=INKEY$:S$="":CU=32
12    CU=168-CU:POKELO,CU:FORJJ=1TO20:I$=INKEY$:
      IFI$=""THENNEXTJJ:GOTO12ELSEJJ=20:NEXTJJ:
      I=ASC(I$):IFI=8ANDS$<>""THENPOKELO,32:LO=LO-1:
```

```
        S$=LEFT$(S$,LEN(S$)-1):GOTO12
13      IFI=13ANDS$<>""THENPOKELO,32:PRINT:RETURN
        ELSEIFI<LV ORI>HV THEN12ELSEIFLO-L1=LE THEN12
        ELSEPOKELO,I:S$=S$+I$:LO=LO+1:CU=32:GOTO12
14      '
```

Main body of the program

```
15      CLEAR 200: DIM NU(10):' INITIALIZE VARIABLES
20      GOSUB 100 'PRINT HEADING
25      GOSUB 200 'ACCEPT PARAMETERS AT KEYBOARD
30      GOSUB 300 'ACCEPT DATA AND GRAPH TITLE AT KEYBOARD
35      GOSUB 400 'CHECK PRINTER AND LPRINT TITLE
40      GOSUB 500 'LPRINT Y AXIS AND LABELS AND DATA
45      GOSUB 600 'LPRINT X AXIS AND LABELS
50      GOSUB 700 'AWAIT KEYPRESS
55      GOTO 15
99 '
```

Print Heading

```
100     CLS: PRINT@16,STRING$(32,140): PRINT @ 83,
        "Printing Bar Graph Program"
110     PRINT @ 148, "Written by Dennis Tanner":
        PRINT @ 208, STRING$(32,140)
120     RETURN
199     '
```

Accept Parameters at Keyboard

```
200     PRINT @ 341, "X Interval? (1 - 10):" CHR$(30);: LE=2: LV=47:
        HV=58: GOSUB 11: XI=VAL(S$): IF XI<1 OR XI>10 THEN 200
210     PRINT @ 405, "Y Interval? (1 - 10):" CHR$(30);: GOSUB 11:
        YI=VAL(S$): IF YI<1 OR YI>10 THEN 210
220     RETURN
299     '
```

Accept the Data at Keyboard

```
300     PRINT @ 972, "Type in numbers to graph, or 999 to end.";
310     FOR CO=1 TO 100
320       PRINT @ 538, "Number";
330       PRINT @ 602, CO ": " CHR$(30);: LE=3: GOSUB 11: X1=VAL(S$):
          IF X1=999 THEN EN=CO-1: CO=100: NEXT CO: GOTO 360
              ELSE IF X1>10*XI THEN 330
340       NU(INT(X1/XI)) = NU(INT(X1/XI)) + 1
350     NEXT CO: EN=100
360     PRINT @ 536, CHR$(31); "Title for graph: ": LE=63: LV=32: HV=127
370     GOSUB 11: TI$=S$: RETURN
399     '
```

LPRINT Title

```
400     CLS: PRINT @ 76, "Press <ENTER> when the printer is ready.";
410     I$=INKEY$
420     I$=INKEY$: IF I$<>CHR$(13) THEN 420
```

```
430   LPRINT TAB(40-LEN(TI$)/2); TI$
440   RETURN
499   '
      LPRINT Y Axis and Labels, and Data

500   LPRINT " ": PRINT @ 132, "Printing line ";
510   FOR Y = 10 TO 1 STEP -1
520     PRINT Y;
530     LPRINT TAB(5); YI*Y; TAB(10); "!";
540     FOR X = 0 TO 10
550       IF NU(X) >= Y*YI
              THEN LPRINT "     *";
              ELSE LPRINT "      ";
560     NEXT X: LPRINT " ": LPRINT " "
570   NEXT Y
580   RETURN
599   '
      LPRINT X Axis and Labels

600   LPRINT TAB(9); STRING$(69,"-"): LPRINT " "
610   FOR J=0 TO 10
620     LPRINT TAB(15+6*J); J*XI;
630   NEXT J
640   LPRINT " "
650   RETURN
699   '
      Await Keypress

700   PRINT @ 980, "Press any key to begin again.";
710   I$=INKEY$
720   I$=INKEY$: IF I$="" THEN 710
730   RETURN
```

Listing 5-C

Line 5 "jumps" around the keyboard input routine.

Lines 10 to 13 contain the keyboard input routine from Chapter 3.

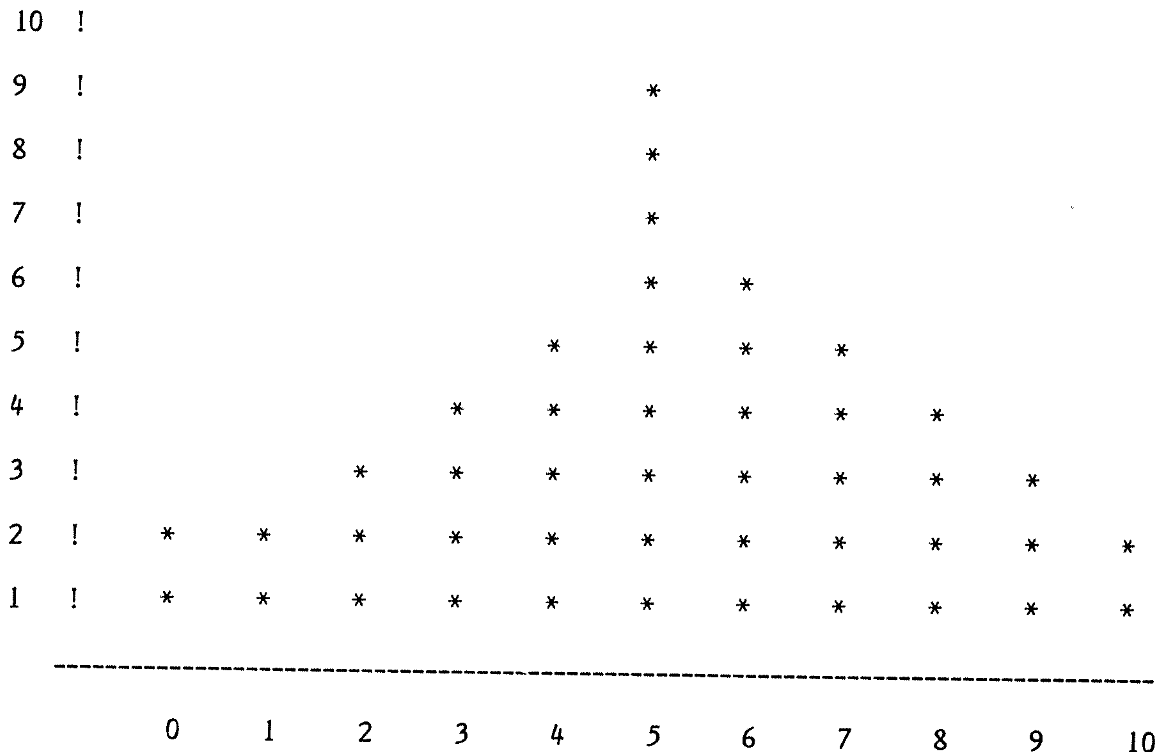
Lines 15 to 55 constitute the main body of the program. Note that there is no branching in the main body of the program, so there is only one execution path in the program.

Lines 100 to 120 print the heading. The STRING$ function is used to display graphics around the printed heading.

Lines 200 to 220 accept the X and Y intervals with values between 1 and 10. These values are stored in the variables XI and YI respectively, and are used in later calculations.

Lines 300 to 370 accept the values to be graphed at the keyboard.

Sample Printout for Program in Listing 5-C



Printout 5-C

Line 340 does the actual counting of the points by incrementing the proper element of the NU() array. To determine which element to increment, the accepted number X1 is divided by the X increment XI, thus putting X1 on the proper scale.

Lines 400 to 440 give the user a chance to get the printer ready, and print the title on the printer.

Lines 500 to 580 do the actual LPRINTing. This section is important and complex enough to warrant a line-by-line anaylsis.

Line	Description
500	LPRINTs one linefeed (" "), then PRINTs the message on the video.
510	Begins a loop on the Y variable. Since the printer starts at the top, the program must start at the top of the numbers. It therefore goes from 10 to 1 by steps of -1.
520	The Y value is PRINTed on the video display so the user can easily

keep track of which line is being LPRINTed.

530 Each time through the Y loop produces one horizontal line on the printer. So this line LPRINTs the first TAB, the Y coordinate (YI * Y), another TAB, and an exclamation mark which serves as a boundary.

540 The X loop goes from 0 to 10 on each line of the printout.

550 If the counter NU(X) for the current X value is as large as or larger than the current Y value, the program LPRINTs an asterisk. Otherwise, it LPRINTs a space. Both the asterisk and the space are padded with extra spaces to spread them out over the X axis.

560 The X loop ends. One linefeed (" ") goes to a new line, and the next one skips an extra line.

570 The Y loop ends.

580 The subroutine ends.

Lines 600 to 650 LPRINT a STRING$ of hyphens and print the X coordinates.

Lines 700 to 720 await a keypress before the program begins again.

As the loop in lines 500 to 580 shows, an unique characteristic of printer graphics is that often **more calculations must be done because all printing is done from the top down, with no provision for going back.**

PRINTING BANNERS WITH THE PRINTER

One of the more useful and entertaining uses of printer graphics is printing large-lettered banners. The program in Listing 5-D does just that. In this program, each capital letter has a string of characters as an element in a string array that describes it. Each letter is defined in a 5 x 7 matrix as like this:

```
-----------------------------------------
!   7   !  14   !  21   !  28   !X 35  X!
-----------------------------------------
!   6   !  13   !  20   !  27   !X 34  X!
-----------------------------------------
!   5   !  12   !  19   !  26   !X 33  X!
-----------------------------------------
!   4   !  11   !  18   !  25   !X 32  X!
-----------------------------------------
!X  3  X!  10   !  17   !  24   !X 31  X!
-----------------------------------------
!X  2  X!   9   !  16   !  23   !X 30  X!
-----------------------------------------
!X  1  X!X  8  X!X 15  X!X 22  X!X 29  X!
-----------------------------------------
```

Figure 5-1

Notice that the matrix is seven characters high and five characters across. The numbers go from bottom to top, and from left to right.

Several of the numbered cells (the 1, 2, and 3 for example) in the matrix have X's in them. If you look carefully at these X's, you'll see that they form the shape of a capital J. The data string in Listing 5-D (line 118) that describes the capital J is as follows:

```
118 S$(10) = "11100001000000100000010000001111111" 'J
```

The first three characters in the string are 1's, and the corresponding positions in the matrix (1, 2, and 3) have X's. The next four characters in S$(10) are 0's, and the corresponding positions in the matrix (4, 5, 6, and 7) contain no X's. The eighth character in the string is a 1, and position eight in the matrix contains an X. In this manner, **the entire matrix containing the "J" is described by the string S$(10).**

Each letter of the alphabet and several punctuation marks have corresponding strings in the S$() array in Listing 5-D.

With that in mind, type in the program. A section-by-section description follows the listing.

```
5      'GOTO 15 'Banner Printout Program by Dennis Tanner
10     '
       Lines 11 to 13 contain the keyboard input routine
       from Chapter 3.

11     LO=PEEK(16416)+PEEK(16417)*256:L1=LO:I$=INKEY$:S$="":CU=32
12     CU=168-CU:POKELO,CU:FORJJ=1TO20:I$=INKEY$:
       IFI$=""THENNEXTJJ:GOTO12ELSEJJ=20:NEXTJJ:
       I=ASC(I$):IFI=8ANDS$<>""THENPOKELO,32:LO=LO-1:
       S$=LEFT$(S$,LEN(S$)-1):GOTO12
13     IFI=13ANDS$<>""THENPOKELO,32:PRINT:RETURN
       ELSEIFI<LV ORI>HV THEN12ELSEIFLO-L1=LE THEN12
       ELSEPOKELO,I:S$=S$+I$:LO=LO+1:CU=32:GOTO12
14     '
       Main body of the program

15     CLEAR 500: DIM S$(63), YN(35) 'DIMENSION VARIABLES
20     GOSUB 100 'ASSIGN VALUES TO STRING VARIABLES
25     GOSUB 200 'PRINT HEADING
30     GOSUB 300 'INPUT MESSAGE AT KEYBOARD
35     GOSUB 400 'PRINT BANNER ON PRINTER
40     GOTO 25
99     '
       Assign Values to String Variables

100    S$(1)="11111000001010000100100010101111100" 'A
102    S$(2)="11111111001001100100110010010110110" 'B
104    S$(3)="11111111000001100000110000011100011" 'C
```

```
106   S$(4)="11111111000001100000110000010111110" 'D
108   S$(5)="11111111001001100100110010011000001" 'E
110   S$(6)="11111110001001000100100010010000001" 'F
112   S$(7)="11111111000001100000110010011111001" 'G
114   S$(8)="11111110001000000100000010001111111" 'H
116   S$(9)="00000001000001111111110000010000000" 'I
118   S$(10)="11100001000000100000010000001111111" 'J
120   S$(11)="11111110001000001010001000101000001" 'K
122   S$(12)="11111111000000100000010000001000000" 'L
124   S$(13)="11111110000010000010000000101111111" 'M
126   S$(14)="11111110000010000010000010001111111" 'N
128   S$(15)="11111111000001100000110000011111111" 'O
130   S$(16)="11111110001001000100100010010000110" 'P
132   S$(17)="11111111000001101000111111110100000" 'Q
134   S$(18)="11111110001001001100101010011000110" 'R
136   S$(19)="01001101001001100100110010010110010" 'S
138   S$(20)="00000010000001111111100000010000001" 'T
140   S$(21)="11111111000000100000010000001111111" 'U
142   S$(22)="00011110110000100000001100000001111" 'V
144   S$(23)="00111111100000001100011000000011111" 'W
146   S$(24)="11000110010100000100000101001100011" 'X
148   S$(25)="00000110000100111100000001000000011" 'Y
150   S$(26)="11000011010001100100110001011000011" 'Z
152   S$(32)="00000000000000000000000000000000000" 'SPACE
154   S$(33)="00000000000000101111100000000000000" '!
156   S$(34)="00000000000011000000000000110000000" 'QUOTES
158   S$(44)="00000001000000010000000000000000000" 'COMMA
160   S$(46)="00000000000000100000000000000000000" 'PERIOD
162   RETURN
199   '
      Print Heading

200   CLS
210   PRINT @ 20, STRING$(23,140)
220   PRINT @ 148, "Banner Printout Program"
230   PRINT @ 276, STRING$(23,131)
240   RETURN
299   '
      Accept Message at Keyboard

300   PRINT @ 448, "Message to be printed:"
310   PRINT
320   LE=63: LV=32: HV=90: GOSUB 11
330   ME$=S$
340   RETURN
399   '
      Print Banner on Printer

400   PRINT @ 980, "< < < Please Wait > > >";
405   L=LEN(ME$)
410   FOR N=1 TO L
```

```
415     Q= ASC(MID$(ME$,N,1)):
        IF Q > 64
          THEN Q = Q - 64
420     FOR I=1 TO 35: YN(I)= VAL(MID$(S$(Q),I,1)): NEXT I
425     LPRINT TAB(10);
430     FOR I=1 TO 5
435       FOR M=1 TO 2
440         FOR J=1 TO 7
445           IF YN((I-1)*7+J) = 1
                THEN LPRINT "XXXXXXX";
                ELSE LPRINT "       ";
450         NEXT J
455         IF M=1
              THEN LPRINT " ": LPRINT TAB(10);
460       NEXT M
465       LPRINT " ": LPRINT TAB(10);
470     NEXT I
475     LPRINT" ": LPRINT" ": LPRINT" ": LPRINT TAB(10);
480   NEXT N
485   RETURN
```

Listing 5-D

Line 5 causes the program to "jump" around the keyboard input routine.

Lines 11 to 13 contain the keyboard input routine from Chapter 3.

Lines 15 to 40 comprise the main body of the program. Notice that the S$() and YN() arrays are dimensioned in line 15.

Lines 100 to 162 assign the character-describing strings to the string variables in the S$() array.

Lines 200 to 240 print the heading with some graphics.

Lines 300 to 340 accept at the keyboard the message to be displayed.

Lines 400 to 485 LPRINT the banner on the printer. Here are some notes on this subroutine:

+ The L loop starts at 1 and goes to the length of the message.
+ Q contains the ASCII value of the current character in the message if the value is less than 65, and the ASCII value - 64 otherwise.
+ The looping variables I and J are used in line 445 to determine whether to LPRINT X's or spaces. Since each letter is printed on its right side, the letters are printed bottom to top, and left to right under the control of these loops.

So far we have seen that the information that determines what will

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX


XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX


                     XXXXXXXXXXXXXXXXXXXXXXXXXXX
                     XXXXXXXXXXXXXXXXXXXXXXXXXXX
       XXXXXXXXXXXXXX
       XXXXXXXXXXXXXX
XXXXXXX
XXXXXXX
       XXXXXXXXXXXXXX
       XXXXXXXXXXXXXX
                     XXXXXXXXXXXXXXXXXXXXXXXXXXX
                     XXXXXXXXXXXXXXXXXXXXXXXXXXX


XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXX              XXXXXXX             XXXXXXX
XXXXXXX              XXXXXXX             XXXXXXX
XXXXXXX              XXXXXXX             XXXXXXX
XXXXXXX              XXXXXXX             XXXXXXX
XXXXXXX              XXXXXXX             XXXXXXX
XXXXXXX              XXXXXXX             XXXXXXX
XXXXXXX                                  XXXXXXX
XXXXXXX                                  XXXXXXX
```

Printout 5-D

be LPRINTed can be stored as text strings in program lines (Listing 5-A), can be calculated with a mathematical function (Listing 5-B), and can be stored in a numeric array (Listing 5-C) or a string array (Listing 5-D). The next two programs determine what to print by reading information directly from the TRS-80 screen.

A SCREEN-PRINTING SUBROUTINE: TEXT INFORMATION ONLY

Sometimes the user wants to "dump" all the information from the video display to the printer. An easy way to do this from the keyboard (on a Model III, or in the Model III mode of the Model 4) is to hold the shift and down-arrow keys and press the "*" key. All the text on the screen will then print to the printer (unless the program running has its own keyboard driver program, which few do).

Listing 5-E shows a subroutine that can be used to accomplish a similar function dynamically, as the program is running. This subroutine can be attached onto the end of other programs and "called" to provide this screen-print capability.

```
10000 ' SCREEN PRINTING SUBROUTINE
        (Letters and numbers; no graphics)
10010 FOR RO=0 TO 15
10020   FOR PO=0 TO 63
10030     CP = PEEK(15360 + RO*64 + PO)
10040     IF CP>31 AND CP<123
            THEN LPRINT CHR$(CP);
            ELSE LPRINT " ";
10050   NEXT PO: LPRINT " "
10060 NEXT RO
10070 RETURN
```

Listing 5-E

```
LIST 10010-10060
10010 FOR RO=0 TO 15
10020  FOR PO=0 TO 63
10030    CP=PEEK( 15360 + RO*64 +PO)
10040    IF CP>31 AND CP<123
          THEN LPRINT CHR$(CP);
          ELSE LPRINT " ";
10050  NEXT PO: LPRINT " "
10060 NEXT RO
READY
>RUN
```

Screen Printout 5-E

The RO loop counts the horizontal rows on the screen, and the PO loop tracks the position within the row. Line 10030 PEEKs the value of the video memory and assigns it to CP. If it is within the acceptable range, line 10040 LPRINTs the character.

A SCREEN PRINTING PROGRAM: PRINTING GRAPHICS FROM THE SCREEN

Have you ever wanted to draw a picture and see it printed out with the printer? This program allows you to do that and more.

Since the video screen is an easily manipulated means of drawing, it is used in this program for designing the picture. A screen editor allows the user to see the image as it is being created. Then the user is given the unusual option of printing the image upright or **on its side.** Then the printer LPRINTs the picture.

```
5     GOTO 14 'Screen Printing Program
              (Graphics Characters)
10
```

Lines 11 to 13 are the INKEY$ routine from Chapter 3.
(The extra PRINT statement in line 13 has been removed.)

```
11    LO=PEEK(16416)+PEEK(16417)*256:L1=LO:I$=INKEY$:S$="":CU=32
12    CU=168-CU:POKELO,CU:FORJJ=1TO20:I$=INKEY$:
      IFI$=""THENNEXTJJ:GOTO12ELSEJJ=20:NEXTJJ:
      I=ASC(I$):IFI=8ANDS$<>""THENPOKELO,32:LO=LO-1:
      S$=LEFT$(S$,LEN(S$)-1):GOTO12
13    IFI=13ANDS$<>""THENPOKELO,32:RETURN
      ELSEIFI<LV ORI>HV THEN12ELSEIFLO-L1=LE THEN12
      ELSEPOKELO,I:S$=S$+I$:LO=LO+1:CU=32:GOTO12
14    CLS'
```

Main body of program

```
15    CLEAR 200: DEFINT A-Z: 'INITIALIZE VARIABLES
20    GOSUB 100 'CREATE GRAPHIC ON SCREEN
25    GOSUB 200 'ACCEPT PARAMETERS FOR PRINTING
30    GOSUB 300 'READ GRAPHICS FROM SCREEN
35    IF PW$ = "U"
        THEN GOSUB 400: GOTO 45 'IF UPRIGHT PRINTING
                                 IS CHOSEN, DO SUB. 400
                                 THEN START OVER
40    GOSUB 500 'OTHERWISE, PRINT SCREEN SIDEWAYS
45    GOTO 15   'START OVER
99    '
```

Create Graphic on Screen

```
100   ON ERROR GOTO 160
105   I$ = INKEY$
110   PRINT @ 960, X; Y; CHR$(30);
```

```
115  I$ = INKEY$: IF I$ = "" THEN 115
120  IF I$ = CHR$(31) THEN
       CLS:
       GOTO110          'CLEAR KEY
125  IF I$ = CHR$(9) THEN
       X=X+1:
       SET(X,Y):
       GOTO 110         'RIGHT ARROW KEY
130  IF I$ = CHR$(8) THEN
       X=X-1:
       SET(X,Y):
       GOTO 110         'LEFT ARROW KEY
135  IF I$ = CHR$(91) THEN
       Y=Y-1:
       SET(X,Y):
       GOTO 110         'UP ARROW KEY
140  IF I$ = CHR$(10) THEN
       Y=Y+1:
       SET(X,Y):
       GOTO 110         'DOWN ARROW KEY
145  IF I$=CHR$(32) THEN
       RESET(X,Y):
       GOTO 110         'SPACE BAR
150  IF I$="Y" OR I$="y" THEN
       ON ERROR GOTO 0:
       RETURN
155  GOTO 110
160  PRINT @ 960, "ERROR...";
165  FOR K=1 TO 400: NEXT K
170  PRINT @ 960, "        ";
175  RESUME NEXT
199  '
     Accept Parameters for Printing

200  PRINT @ 960, "Print (U)pright or (S)ideways? "; CHR$(30);
210  LE=1: LV=65: HV=90: GOSUB 11:
     IF S$<>"U" AND S$<>"S" THEN 200
220  PW$ = S$:
     IF PW$="U"
       THEN PRINT @ 960, "Upright.  ";
       ELSE PRINT @ 960, "Sideways. ";
230  PRINT @ 970, "Maximum X pixel? "; CHR$(30);
240  LE=3:LV=48:HV=57:GOSUB 11: XM=VAL(S$): IF XM > 127 THEN 230
250  PRINT @ 970, CHR$(30); "Max. X="; XM;
260  PRINT @ 985, "Maximum Y pixel? "; CHR$(30);
270  GOSUB 11: YM=VAL(S$): IF YM>44 THEN 260
280  PRINT @ 985, CHR$(30); "Max. Y="; YM;
290  RETURN
299  '
     Read Graphics from Screen
```

```
300   PRINT @ 1001, "<<Please wait>>";
310   DIM Z(XM,YM)
320   FOR X=0 TO XM
330     FOR Y=0 TO YM
340       Z(X,Y) = POINT(X,Y)
350     NEXT Y
360   NEXT X
370   RETURN
399   '
      Print Graphic Upright

400   PRINT @ 1001, "Printing........";
410   FOR Y=0 TO YM
420     FOR X=0 TO XM
430       IF Z(X,Y) = -1
              THEN LPRINT "X";
              ELSE LPRINT " ";
440     NEXT X: LPRINT " "
450   NEXT Y
460   RETURN
499   '
      Print Screen Sideways
500   PRINT @ 1001, "Printing........";
510   FOR X=0 TO XM
520     FOR Y=YM TO 0 STEP -1
530       IF Z(X,Y) = -1
              THEN LPRINT "X";
              ELSE LPRINT " ";
540     NEXT Y: LPRINT " "
550   NEXT X
560   RETURN
```

Listing 5-F

The programming techniques in this program are straightforward. Here are some notes:

+ The screen is cleared (line 14) only the first time the program is run. When the program LPRINTs the graphics and starts over, it loops back to line 15 without clearing the screen.

+ Only the first 15 lines of the screen are read from the screen for LPRINTing. The user should be careful not to include the bottom line of the screen in the picture.

+ The parameters are accepted on the bottom line of the screen <u>after</u> the graphics are created. The user must pay attention to the maximum X and Y values reached during the creation process so they can be typed in below the picture.

+ The Z() array is dimensioned immediately before the screen is read.

```
XXXXXXXXXXXXXXXXXXXXXXXXXX
X                        X
X                        X
X                        X
X                        X
X                        X
X                        X
X        XXXXXX          X
X        X    X          X
X        X    X          X
X        X    X          X
XXXXXXXXXX    XXXXXXXXXXXX
```

Printout 5-F

The dimensions are set to the maximum X and Y values the user has selected.

With this program, you can create graphics on the screen, print them out, edit them if desired, and print them out again. The only limitation of this program is your graphics and screen-editing ability!

USING GRAPHICS CHARACTERS ON PRINTERS

Several line printers allow the use of graphics characters on the printers themselves. For example the Radio Shack Line Printer VI allows printing of any part of a four-pixel block (as opposed to the screen, which uses a six-pixel block). This printer also allows the printing of a triangular shape with four different orientations.

Since there are many different kinds of printers with different codes and techniques for graphics programming, this section will look at the Line Printer VI as an example, then address general techniques for graphics programming on printers with special graphics capabilities.

Some printers have special characters that control the size of the characters printed. On the Line Printer VI, for example, LPRINTing a character 31 changes the printer to the double-size mode. LPRINTing character 30 changes back to regular sized characters.

The instruction

```
LPRINT CHR$(27); CHR$(14)
```

switches to condensed characters, and

LPRINT CHR$(27); CHR$(15)

switches back to normal characters.

Another set of software "switches" controls the vertical line spacing on the printer. For example,

LPRINT CHR$(27); CHR$(28)

causes the printer to LPRINT 12 lines per inch.

LPRINT CHR$(27); CHR$(56)

changes to 8 lines per inch, and

LPRINT CHR$(27); CHR$(54)

changes to 6 lines per inch.

These control characters function with a Radio Shack Line Printer VI, but other printers may have different codes for accomplishing the same things. Information on the codes your printer uses is found in the printer owner's manual.

Here are general guidelines for using these printers:

1) Since your printer starts at the top of the page and prints to the bottom, you will have to do all the calculations for the top line before you begin printing. (This is different from the screen, on which you can display information and graphics anywhere at any time. Some printers also have this capability: see your owner's manual.)

2) If you want your printout to be continuous (rather than with some blank space between the lines, as text is printed), you may have to LPRINT some control codes to set the vertical spacing.

3) Some printers overheat if the graphics mode is used continuously, so be sure the check your owner's manual for recommendations on the amount of load your printer head can withstand.

4) If you are writing programs to distribute to others, check the printer codes of various printers, and specify in the documentation which you supply with your program, which printers the graphics will work with.

Chapter Summary

The **LPRINT** instruction prints text characters on a printer the same way the **PRINT** instruction displays text characters on the video screen.

Instructions such as TAB, the comma, and the semicolon which can be used with **PRINT** to control forward movement of the cursor position on the video display, can also be used to control positioning of text on a printer with **LPRINT**.

Instructions such as PRINT @, POKE (for the video display), and SET/RESET/POINT, which can be used with **PRINT** control position of text and graphics anywhere on the screen, cannot be used with the printer.

Simple pictures may be created on the printer using **LPRINT** with text in quotation marks.

Graphics may be created on the printer using **LPRINT TAB** with numbers mathematically calculated or otherwise generated or stored.

If calculations or computations must be done to determine placement of characters on the screen (such as in our bar graph program), this processing must be finished before LPRINTing begins.

Information in a string array can be used to determine information to LPRINT (as in our graphics program). The **MID$()** function is used in this case to analyze the strings.

Text characters can be read directly from the video memory to facilitate duplicating all or part of the text screen to the printer.

Graphics characters can be read from the screen using the POINT instruction. This can facilitate duplication of all or part of a graphics screen on the printer using text characters.

Some line printers permit special graphics printing using special codes or other instructions. You should check your printer owner's manual for the specifics of these instructions.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Write a program that prints a simple picture on a printer, using a series of LPRINT instructions. The data for this picture will be literal information stored in text strings.

B. Write a program that draws a sine wave on a printer. The TAB function used with mathematical calculations will be used to determine the print locations.

C. Write a program that draws a graph on the printer. The data determining the LPRINT locations in this program will be stored in a numeric array.

D. Write a program that prints one large graphic letter or number on a printer. String arrays will store the data for this program.

E. Write a subroutine that prints all the text characters (but not the graphics characters) from the TRS-80 screen to the keyboard. The text characters on the screen will determine the graphics in this program.

F. Write a program that reads the graphics from a portion of the screen and prints them on a printer. The graphics characters on the screen will determine what's printed in this program.

More Programming Practice

1. Using simple LPRINT instructions, write a program that prints a likeness of a TRS-80 microcomputer on the printer.

2. Write a program that reads a math function (such as X=Y*Y) from a program line and prints a graph of the function using the TAB instruction.

3. Write a program that READs DATA statements, and using the data gathered, LPRINTs a graph with a child's age on the X axis, and the child's height on the Y axis.

4. Design a matrix that describes the numerals from 0 to 9, and write a program that LPRINTs specified numbers on the printer.

5. Write a program that uses the POINT instruction to read <u>every</u> graphics point on the screen and send it to a printer.

Chapter Checkup

1. In the program in Listing 5-A, why can lines 10 (CLS) and 150 (GOTO 150) be omitted without affecting the printout?

2. What changes would you make to the program in Listing 5-C to print a centered graph on a 132 column printer?

3. In what format are the numbers in line 520 of Listing 5-C PRINTed on the screen?

4. In the program in Listing 5-C, how is the LPRINT location of the title of the graph determined?

5. In the program in Listing 5-D, what function does the M loop in lines 435 and 460 perform?

6. In the program in Listing 5-E, show the calculations and values

used in lines 10010 to 10030 to determine the minimum and maximum values of the video memory.

7. In the program in Listing 5-E, what happens when the program reads a screen character that is not a text character?

8. In the program in Listing 5-F, why has the PRINT instruction been taken out of line 13?

9. In the program in Listing 5-F, what are the major differences between the subroutine beginning in line 400 and the one beginning in line 500?

10. In the program in Listing 5-F, what is the advantage of not clearing the screen each time the program loops back to start over (see lines 14, 15 and 45)?

Chapter 6.
Using the Variable Pointer
To Create Graphics

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write the description of the VARPTR instruction (as used with string variables).
B. For the VARPTR of a string variable, tell what is defined by each of the three bytes starting at the VARPTR location.
C. Given the three bytes starting at the VARPTR location, determine the memory location of the characters of the string variable.
D. Write a program that allows the user to view and edit the contents of the memory locations containing the characters of a string stored in a variable.

INTRODUCTION

Most of the BASIC instructions that allow manipulation of graphics on the TRS-80 microcomputers are direct graphics instructions: SET, RESET, POINT, PRINT, and LPRINT. Some others let you manipulate video memory directly: POKE and PEEK. The instruction we will examine in this chapter shows us where certain information about string variables is stored in memory. This information can be accessed for graphics design or other purposes.

THE VARPTR INSTRUCTION

The BASIC instruction VARPTR (pronounced "Variable Pointer") allows the user to determine where the raw data assigned to a variable is stored in the computer's memory. Thus, the VARPTR points to the variable.

Each kind of variable (integer, single-precision, double-precision, string, and array) has a VARPTR with a unique format. We will limit the discussion in this chapter to VARPTRs of string variables, the kind that store graphics.

First let's build a program that allows us to manipulate the characters in string variables using the VARPTR instruction.

```
10 CLS
20 A$= "THIS IS THE TEST!"
30 PRINT "A$= "; A$
40 VA= VARPTR(A$)
```

Listing 6-A

Lines 10 to 30 are straightforward. Line 40 shows us the VARPTR instruction in its correct format. VARPTR is used in this program as a function of A$, in the same format as LEN(A$). The number representing

the VARPTR of A$ is assigned to the numeric variable VA.

```
10 CLS
20 A$= "THIS IS THE TEST!"
30 PRINT "A$= "; A$
40 VA= VARPTR(A$):
       IF VA>2 15
         THEN VA= VA-2 16
50 PRINT "THE VARIABLE POINTER OF A$ IS" VA
```

Listing 6-B

The PEEK instruction (which will appear later in the program) can access only numbers between -32768 and 32767. The statements added to line 40 check for the value of VA. If this value is greater than 2 15 (or 32768), it cannot be accessed by the PEEK instruction, so it must be changed to its corresponding **negative** number. This is accomplished by subtracting 2 16 (or 65536) from the number.

If VA is initially 35000 (which is greater than 32768), it will be changed to 35000-65536, or -30536. To POKE to or PEEK from location 35000, we must use the number -30536 instead. Line 40 checks the size of VA and adjusts it if needed.

Line 50 PRINTs the value of the variable VA, the appropriately adjusted variable pointer.

```
10 CLS
20 A$= "THIS IS THE TEST!"
30 PRINT "A$= "; A$
40 VA= VARPTR(A$):
       IF VA>2 15
         THEN VA= VA-2 16
50 PRINT "THE VARIABLE POINTER OF A$ IS" VA
60 V0= PEEK(VA)
70 PRINT "PEEK(VARIABLE POINTER)=" V0
80 PRINT "THE LENGTH OF A$=" V0
```

Listing 6-C

The VARPTR instruction returns our number VA, which we use as the address of a memory location. If we PEEK the value stored in that memory location, we find a number representing the length of the string variable A$. Aha! Now we're getting somewhere.

The memory location pointed to by the VARPTR contains the length of the string stored in the string variable.

```
10 CLS
20 A$= "THIS IS THE TEST!"
30 PRINT "A$= "; A$
```

```
40 VA= VARPTR(A$):
     IF VA>2 15
       THEN VA= VA-2 16
50 PRINT "THE VARIABLE POINTER OF A$ IS" VA
60 V0= PEEK(VA)
70 PRINT "PEEK(VARIABLE POINTER)=" V0
80 PRINT "THE LENGTH OF A$=" V0
90 V1= PEEK(VA+1)
100 PRINT "PEEK(VARIABLE POINTER+1)=" V1
110 V2= PEEK(VA+2)
120 PRINT "PEEK(VARIABLE POINTER+2)=" V2
130 SL= V1 + V2*256
140 PRINT "THE VARIABLE CHARACTERS ARE
    STORED AT"
150 PRINT V1 "+" V2 "*" 256 "=" SL
```

Listing 6-D

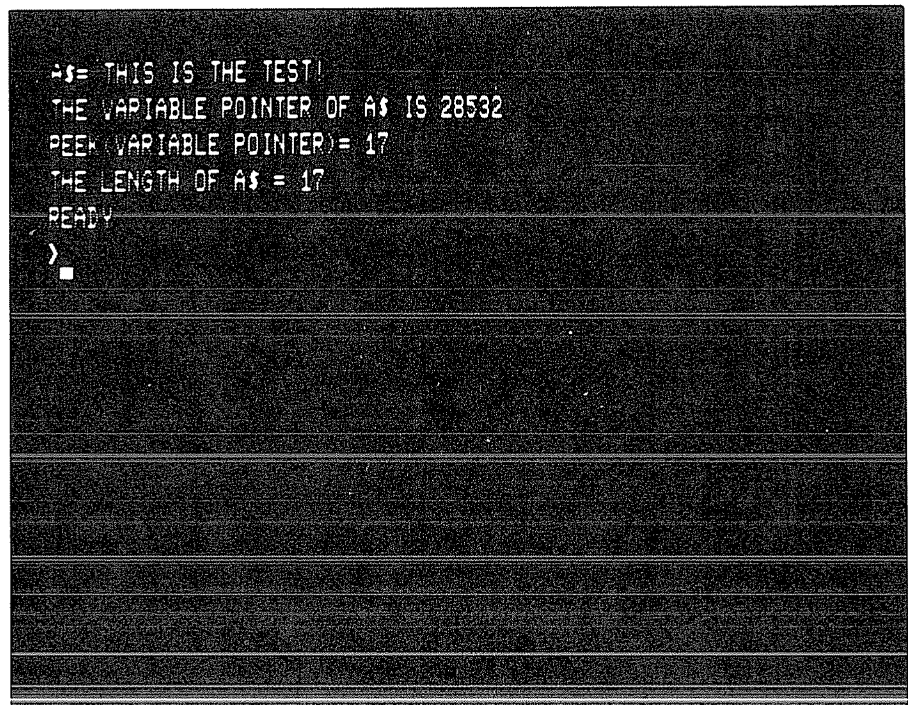


Photo 6-C

Let's take moment to look at a chart outlining this information before getting into a full explanation.

V0 is the VARPTR of A$.

V0	V1	V2
Length of A$	Least Significant Byte of Storage Location	Most Significant Byte of Storage Location

Figure 6-1

```
A$= THIS IS THE TEST!
THE VARIABLE POINTER OF A$ IS 28746
PEEK(VARIABLE POINTER)= 17
THE LENGTH OF A$ = 17
PEEK(VARIABLE POINTER+1)= 139
PEEK(VARIABLE POINTER+2)= 110
THE VARIABLE CHARACTERS ARE STORED AT
 139 + 110 * 256 = 28299
READY
>
```

Photo 6-D

V0, V1, and V2 represent three adjacent memory locations. As we have already seen, V0 contains the length of the string. The numbers in V1 and V2 can be used together to determine the location of the string data itself. Look back at line 130. In this calculation the number in V2 is multiplied by 256, so it is called the Most Significant Byte. The number in V1 is used without multiplying, so it is called the Least Significant Byte. The formula

V1 + V2*256

gives us the address of the memory location we are looking for. This value is assigned to the variable SL and is the storage location of the beginning of the data contained in the variable.

The two memory locations immediately following the variable pointer location contain numbers that point to the raw data contained in the string variable.

```
160 INPUT"READY TO VIEW CHARACTERS"; J$
170 CLS
180 FOR J=0 TO V0-1
190  PRINT "PEEK(" SL+J ")=" PEEK(SL+J)
     "=" CHR$(PEEK(SL+J))
200 NEXT J
```

Listing 6-E

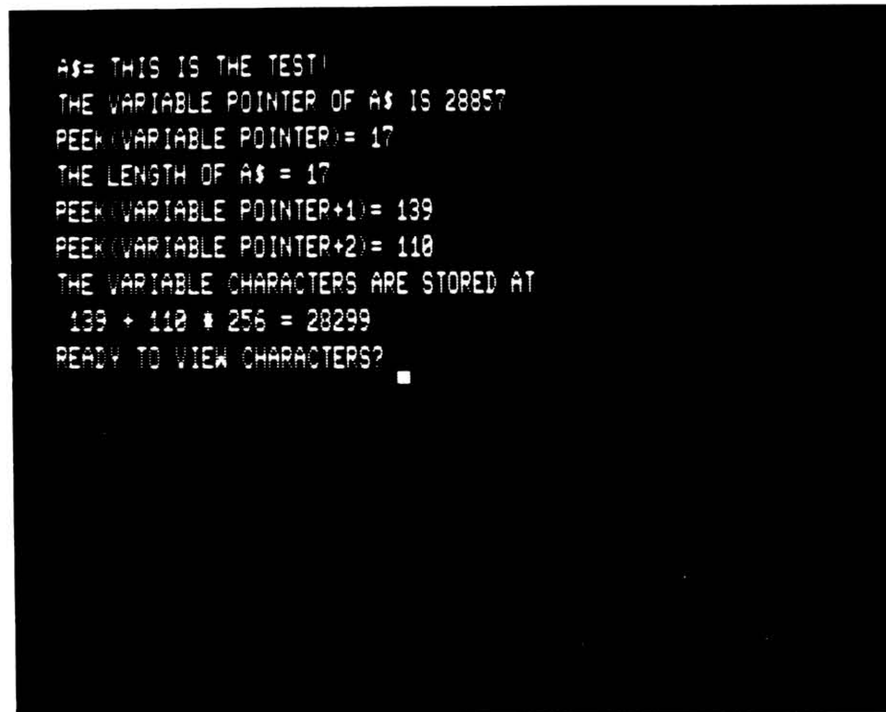


Photo 6-E-1

This next secton of the program displays the contents of the string.

Line	Description
160	Holds previous display until <ENTER> is pressed.

170 Clears the screen.

180 Starts the J FOR-NEXT loop. Since V0 contains the length of the string, a loop from 1 to V0 would count up to the length of the string. Similarly, a loop from 0 to V0-1 also counts up to the length of the string. This latter form is preferable since we will add the J value to the beginning memory location, and we want to start by adding 0.

190 This line PRINTs
- the word "PEEK("
- the memory location address SL+J
- the characters ")="
- the number stored at address SL+J
- the equals sign
- the character represented by the number stored in the address SL+J

If our address is 31000 and it contains the value 65, this line will PRINT

PEEK(31000)= 65 = A.

As this is repeated through the loop, the user will see the contents of the entire string pointed to by the VARPTR.

200 The J loop ends.

```
PEEK( 28301 )= 73 =I
PEEK( 28302 )= 83 =S
PEEK( 28303 )= 32 =
PEEK( 28304 )= 73 =I
PEEK( 28305 )= 83 =S
PEEK( 28306 )= 32 =
PEEK( 28307 )= 84 =T
PEEK( 28308 )= 72 =H
PEEK( 28309 )= 69 =E
PEEK( 28310 )= 32 =
PEEK( 28311 )= 84 =T
PEEK( 28312 )= 69 =E
PEEK( 28313 )= 83 =S
PEEK( 28314 )= 84 =T
PEEK( 28315 )= 33 =!
READY TO EDIT CHARACTERS?
```

Photo 6-E-2

So now we have displayed the memory locations and their contents of the contents of the string. This is something entirely new. When we worked with string variables before, we were concerned about their contents but not the memory locations that contained them.

The next section of the program allows the user to edit the characters in the string by changing the values stored in the memory locations containing the string.

```
210 INPUT "READY TO EDIT CHARACTERS"; J$
220 CLS
230 INPUT "START WITH WHICH CHARACTER"; SC
240 FOR J= SC-1 TO V0-1
250  CLS: PRINT "CHARACTER" J+1
260  PRINT "PEEK(" SL+J ")=" PEEK(SL+J)
    "=" CHR$(PEEK(SL+J))
270  INPUT "NEW CHARACTER"; NC
280  POKE SL+J, NC
290 NEXT J
300 LIST 20
```

Listing 6-F

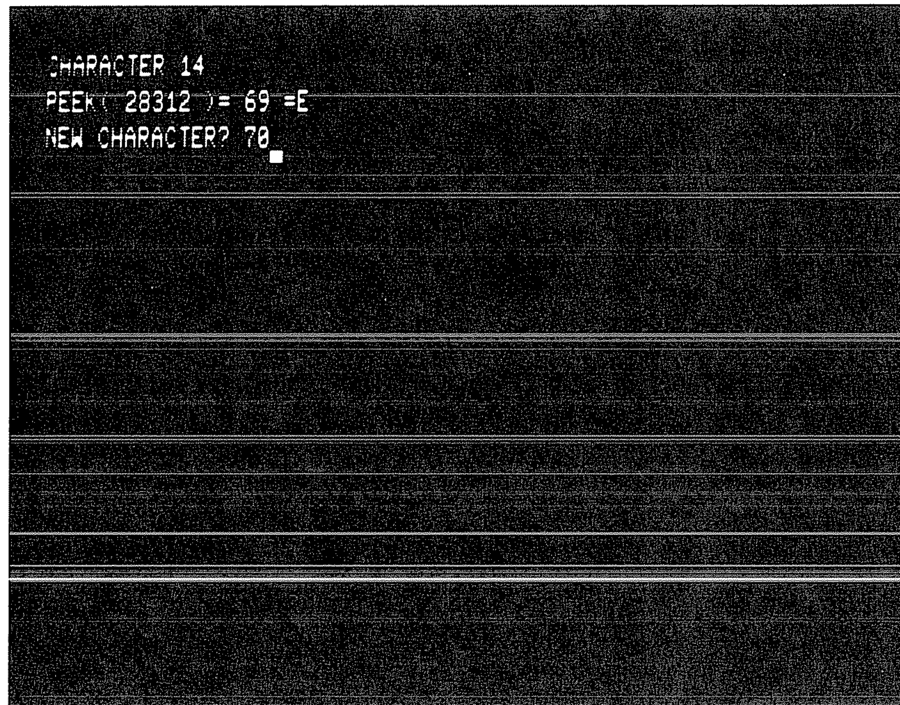


Photo 6-F

Let's examine each line of this section of the program.

Line	Description
210	Holds previous display until <ENTER> is pressed.
220	Clears the screen.
230	Accepts at the keyboard the character to use to start editing. The user would type "1" to start with the first character, "2" to start with the next character, etc.
240	This starts the loop. Once again, 1 is subtracted from the value the user types in. So if the user types "1" the start with the first character, 0 will be added to the value, which is the desired effect.
250	The screen is cleared and the current value of the looping variable J is printed, offset by one to restore it to the user's perspective of the value.
260	This line PRINTs the same information that line 190 above PRINTs.
270	The user types in the value of the new character desired for that memory location. For example, if the character "A" is desired, the user types in the ASCII value "65". If a graphics character is desired, the user types in a value between 128 and 191.
280	The value the user types in in POKEd to the memory location which has the address equal to the beginning storage location plus the looping variable.
290	The loop ends.
300	This is a strange line. It lists line 20 of the program. When you run the program, you will see that not only has the value assigned to the variable changed, but also <u>the program line assigning the value to the variable has also changed.</u>

When a string value is assigned to a string variable, POKEing values into the location of the string changes not only the value of the string variable, but also the program line containing the string variable. This is true unless the value of the variable has been changed in subsequent program lines.

Another way to say this is, the VARPTR points to the area of memory containing the actual program lines where the value of the string variable is assigned.

PRINTING THE GRAPHICS ON THE SCREEN

The program in Listing 6-F let the user change the characters in the string stored in a string variable. The next program lets the user do the same thing with access to any character, in any order, and with the graphic always displayed on the screen.

```
10 CLS
20 A$="THIS IS THE TEST!"
30 VA= VARPTR(A$):
   IF VA > 2 15
    THEN VA = VA - 2 16
```

```
40 V0 = PEEK(VA)
50 V1 = PEEK(VA+1)
60 V2 = PEEK(VA+2)
70 SL = V1 + V2*256
80 CLS: PRINT@0, "LENGTH IS "V0; CHR$(30)
90 PRINT@448, "A$="
100 PRINT @ 512, A$;
110 PRINT @ 64, "WHICH CHARACTER"; CHR$(30);
120 INPUT CP
130 IF CP > V0 THEN 110
140 PRINT "CHARACTER" CP "=" PEEK(SL+CP-1)
150 INPUT "NEW VALUE"; NV
160 POKE SL+CP-1, NV
170 GOTO 80
```

Listing 6-G

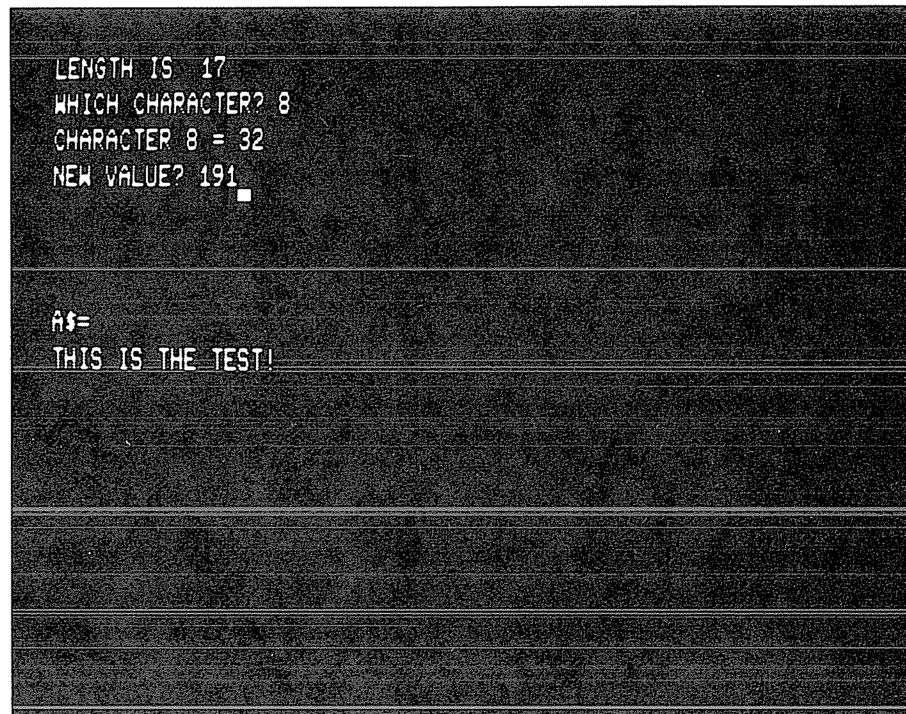


Photo 6-G-1

Lines 10 to 70 assign the value of the string variable A$ (line 20), its VARPTR (line 30), the length of the string (line 40), the variables containing the numbers used to compute the storage location of the string's data (lines 50 and 60), and the storage location (line 70).

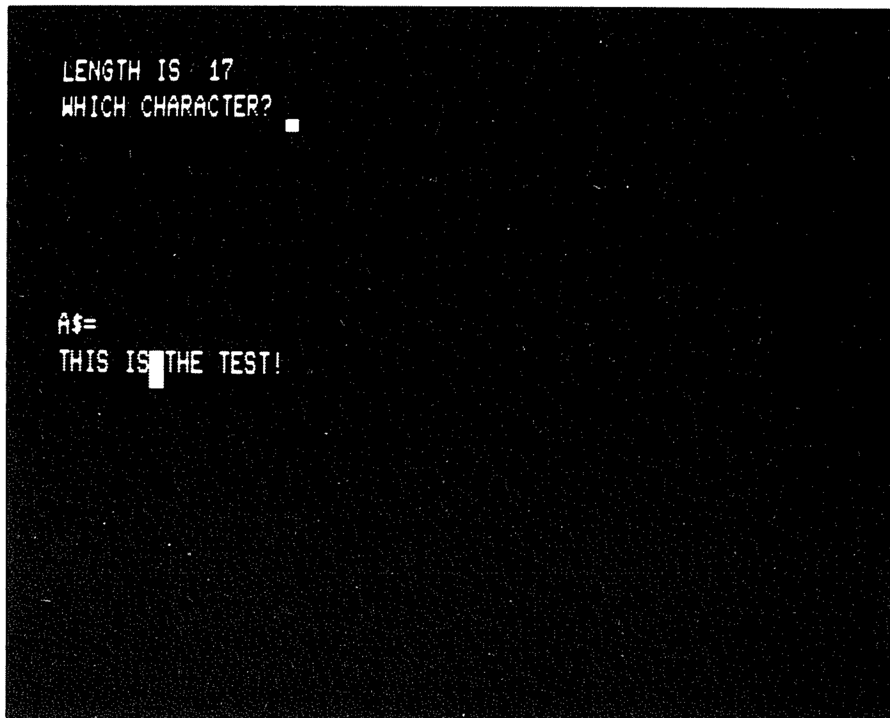


Photo 6-G-2

Let's look at the remainder of the program line by line.

Line	Description
80	Clears the screen. PRINTs the length of the variable (V0) and character 30, which clears the remainder of the line on the display.
90	PRINTs A$= at location 448 (on the left side of the screen, near the vertical center).
100	PRINTs the value of the string A$ on the next line down on the screen.
110	PRINTs at location 64 (near the top of the screen), the question **WHICH CHARACTER** and character 30, which clears the remainder of the line.
120	This INPUT instruction adds the question mark to the "WHICH CHARACTER" question and accepts the user's choice of character at the keyboard. Again,the user would type "1" to choose the first character in the string, "2" to choose the second character, and so on.
130	Prevents the program from using a user-input number beyond the length of the string. (Since line 160 POKEs a value into the location represented by that number, it is important that the location be checked. POKEing around into the wrong part of

memory can have disastrous results.)

140 So the user can be sure the desired memory location is being accessed, the number stored in the memory location is displayed.

150 The user types in the new value (between 0 and 255). This number typed is the number which will be POKEd into the memory location in line 160.

160 The new value specified in POKEd into the memory location specified. This changes the value of the string as it is stored in memory.

170 The program returns for another memory location.

Note that each time the program loops back from line 170 to line 80, the screen is cleared (line 80) and the new value of the string variable is PRINTed (line 100). The user can always see the current value of the variable, and can thus see the effects of the changes made at the keyboard.

This is a short program, but its results are rather dramatic. By changing among text characters (32-127), graphics characters (128-191), and control characters (8, 9, 10, 13, 24-31), the effects of each character on the string can be seen.

REAL-TIME ON-SCREEN EDITING

The two previous programs let the user control what characters a string contained, by typing in the numbers corresponding to the desired characters. In the next program, the VARPTR instruction is then used to determine the location of a string variable in memory. Next the user draws with a screen editor, creating a design right on the screen with the arrow keys. Finally, the program transfers character values from the video memory to the string variable.

```
5     GOTO 15 ' String-Packing Program
10 '
```

Lines 11 to 13 are the INKEY$ routine from Chapter 3 (with the extra PRINT taken out of line 13).

```
11    LO=PEEK(16416)+PEEK(16417)*256:LI=LO:I$=INKEY$:
      S$="":CU=32
12    CU=168-CU:POKELO,CU:FORJJ=1TO20:I$=INKEY$:
      IFI$=""THENNEXTJJ:GOTO12 ELSEJJ=20:NEXTJJ:
      I=ASC(I$):IFI=8ANDS$<>""THENPOKELO,32:LO=LO-1:
      S$=LEFT$(S$,LEN(S$)-1):GOTO12
13    IFI=13ANDS$<>""THENPOKELO,32:RETURN ELSE
      IFI<LV ORI>HV THEN12 ELSEIFLO-LI=LE THEN12
      ELSEPOKELO,I:S$=S$+I$:LO=LO+1:CU=32:GOTO12
14    '
      Main Body of Program

15    CLEAR 200:GOSUB100  'INITIALIZE VARIABLES
20    GOSUB200  'CREATE GRAPHIC ON SCREEN
```

```
25    GOSUB300  'GET PARAMETERS FOR PRINTING
30    GOSUB400  'READ GRAPHICS FROM SCREEN
              AND POKE INTO VARIABLE
35    GOTO15
99    '
      Initialize Variables
100   DEFINT A-Z
110   A$="
                                                  "  '(That's 128
      spaces)'
120   RETURN
199   '
      Create Graphic on Screen

200   CLS: ON ERROR GOTO 260
205   PRINT @0, A$;: I$=INKEY$
210   PRINT @960, X; Y; CHR$(30);
215   I$=INKEY$: IF I$="" THEN 215
220   IF I$=CHR$(31) THEN CLS: GOTO 210
225   IF I$=CHR$(9)
       THEN X=X+1:
            SET (X,Y):
            GOTO 210
230   IF I$=CHR$(8)
       THEN X=X-1:
            SET (X,Y):
            GOTO 210
235   IF I$=CHR$(91)
       THEN Y=Y-1:
            SET (X,Y):
            GOTO 210
240   IF I$=CHR$(10)
       THEN Y=Y+1:
            SET (X,Y):
            GOTO 210
245   IF I$=CHR$(32)
       THEN RESET(X,Y):
            GOTO 210
250   IF I$="Y" OR I$="y"
       THEN ON ERROR GOTO 0:
            RETURN
255   GOTO 215
260   PRINT @ 960, "ERROR...";
265   FOR K=1 TO 400: NEXT K
270   PRINT @ 960, "        ";
275   RESUME NEXT
299   '
      Get Parameters for Printing

300   PRINT @ 960, "LEN(A$)=" LEN(A$);
310   PRINT @ 975, "Maximum X pixel? "; CHR$(30);
```

```
320   LE=3: LV=48: HV=57: GOSUB 11: XM=VAL(S$):
      IF XM>127 THEN 310
330   PRINT @ 975, CHR$(30); "Max. X="; XM;
340   PRINT @ 990, "Maximum Y pixel? "; CHR$(30);
350   GOSUB 11: YM=VAL(S$): IF YM>44 THEN 340
360   PRINT @ 990, CHR$(30); "Max. Y="; YM;
370   RETURN
399   '
      Read Graphics from Screen and POKE into Variable

400   PRINT @ 1005, "<<Please Wait>>";
405   XC = INT((XM+2)/2)
410   YC = INT((YM+3)/3)
415   COUNT = YC*XC + (YC-1)*XC + YC-1
420   IF COUNT > LEN(A$)
       THEN PRINT @ 1005, "String Too Long";:
              PRINT @ 768, " ";:
              END
425   VA=VARPTR(A$): IF VA>2 15 THEN VA=VA-2 16
430   V1=PEEK(VA+1): V2=PEEK(VA+2): SL=V1+V2*256
435   IF SL > 2 15 THEN SL=SL-2 16
440   ST=SL
445   SL=SL-1
450   FOR Y=0 TO YC-1
455     FOR X=0 TO XC-1
460       SL=SL+1
465       POKE SL, PEEK(15360 + X + Y*64)
470     NEXT X
475     SL=SL+1: IF SL-ST >= COUNT THEN POKE SL,32:
        GOTO 510
480     POKE SL, 26
485     FOR Z = 0 TO XC-1
490       SL=SL+1
495       POKE SL,8
500     NEXT Z
505   NEXT Y
510   SL=SL+1: IF SL-ST >= LEN(A$) THEN 525
515   POKE SL,32
520   GOTO 510
525   RETURN
```

Listing 6-H

To use this program, draw your picture with the arrow keys. Pressing the space bar erases the pixel at the current position. As you draw, the current values for X and Y are displayed in the lower left corner of the screen. Keep track of the greatest X value and the greatest Y value you use.

When you have finished your drawing, press the "Y" key (for "Yes"). The program will prompt you to key in your maximum X value and Y

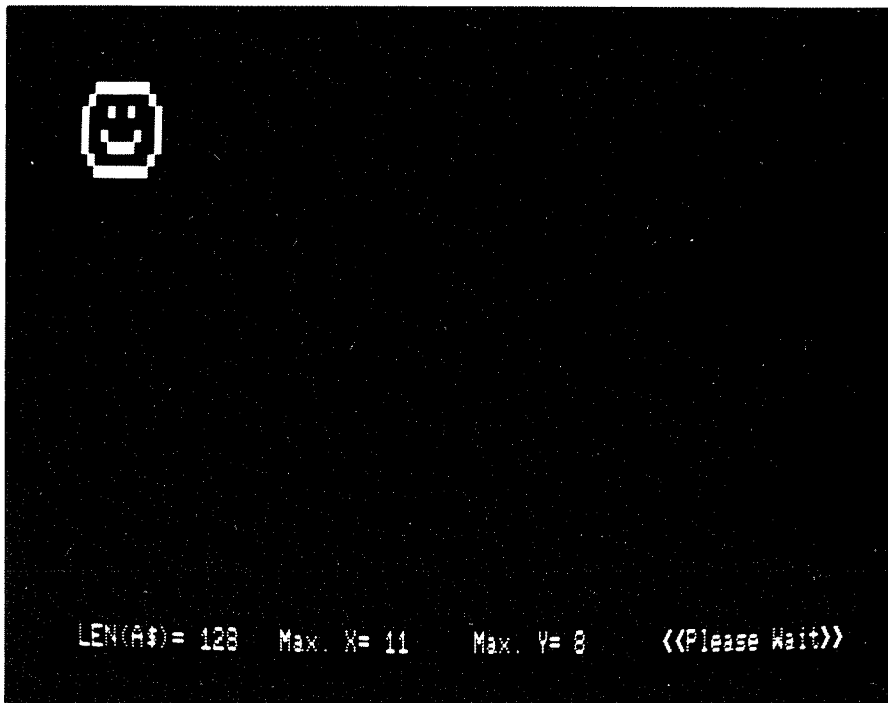


Photo 6-H-1

value, so use the ones you were keeping track of as you drew the picture. The "Please Wait" prompt will appear as the computes the number of characters needed to store your graphic. If your graphic requires more characters than there are in the string, the message "String Too Long" appears on the screen.

If the string will fit, the program reads the video memory and POKEs the appropriate values into the locations where the string data is stored in memory. For the remaining memory locations in the string, it POKEs a 32 (space), thus padding the remainder of the string with spaces. This eliminates any leftover characters that may remain from the previous graphic.

Let's examine each section of this program.

Line 5 contains the program title and jumps around the keyboard-input routine in lines 11 to 13.

Lines 15 to 35 contain the main body of the program. Notice that it is a linear program, with no jumping around in the main body of the program except that line 35 goes back to line 15 when the program

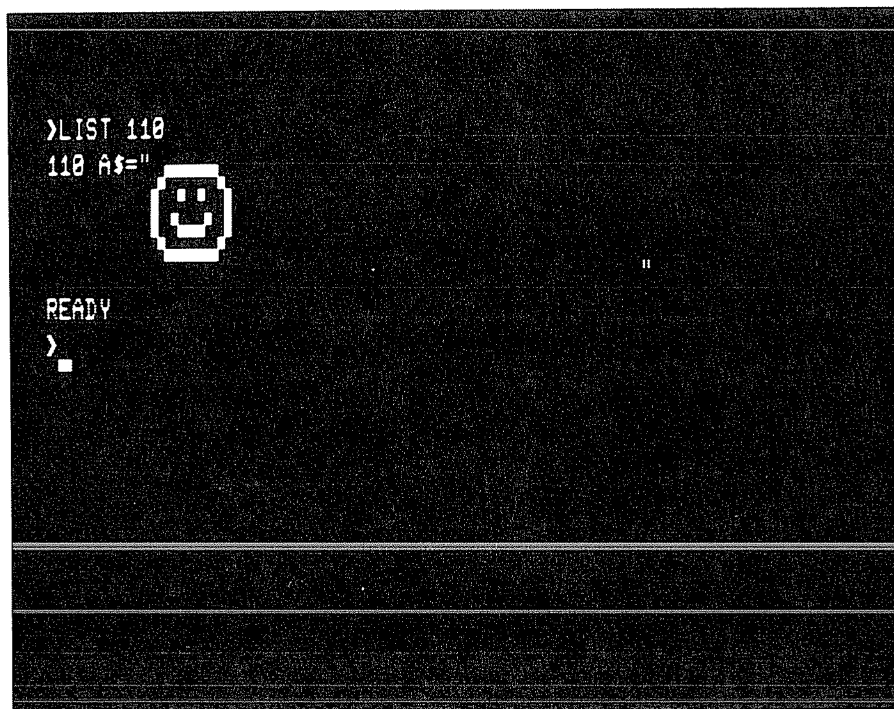


Photo 6-H-2

finishes reading the screen and POKEing it to the variable.

Lines 100 to 120 initialize the variable. All the variables are defined as integers (for faster program execution) in line 100. A$ is assigned its initial value of 128 spaces in line 110. The data to be read later from the screen will be POKEd into these 128 spaces in the program.

Lines 200 to 275 check the keyboard with the INKEY$ instruction in line 215. If an arrow, the clear key, the space bar, or the "Y" or "y" key is pressed, the program responds appropriately.

Lines 300 to 370 ask the user for the maximum X and Y dimensions. The keyboard input is used for the keyboard responses. (If an INPUT instruction were used, pressing ENTER would cause the screen to scroll.) The values are checked to be sure they will fit on the portion of the screen allotted to the graphics, which includes all but the last character line of the screen. If the user puts in values that do not include all the graphic drawn, only the portion of the graphic bordered by the values given will be included in the string being built.

Lines 400 to 525 include the most complex part of this routine. A line-by-line description is appropriate here.

Line	Description
400	PRINTs <<**Please Wait**>> message on the bottom line of the screen. Notice the semicolon after the material to be PRINTed. This semicolon suppress the linefeed after the PRINT, and thus prevents scrolling.
405	Calculates the number of character spaces on the X axis needed for the graphic, based on the number of pixels the user has specified as "X maximum."
410	Calculates the number of character spaces on the Y axis needed for the graphic, based on the number of pixels the user has specified as "Y maximum."
415	Calculates the total count of character spaces needed for the entire graphic. The calculation includes: * YC*XC, the X count times the Y count; * (YC-1)*XC, since every Y row but the last requires a row of BACKSPACE characters (character 8); and * YC-1, since every Y row but the last requires an extra linefeed character (character 26).
420	If the number of characters needed to store the string is greater than the length of the string, the program returns a message and ends. If this were not checked and the program POKEd as many characters as the graphic required, regardless of the length of the string variable in the program, numbers might be POKEd into memory beyond the end of the string. This could be disastrous, since it would POKE characters into the code of the program and surely scramble it to an unexecutable state.
425	Assigns the variable pointer of A$ to the variable VA. Checks for values beyond the PEEKing range and adjusts if needed.
430	Calculates the location of the raw data in memory using the VARPTR values.
435	Once again, if the value is beyond the PEEKing range, it is adjusted.
440	SL is the string location of the raw data in memory. Since SL will be incremented and used as a counter, the variable ST is assigned to SL's starting value for later reference.
445	SL is incremented each time the loop that follows is run, including the first time. So SL is decremented by in line 445 so it will contain its starting value when it is incremented the first time (line 460).
450	The Y loop starts at 0, counting up to YC-1 (one less than the number of characters needed on the Y axis, since it started at zero).
455	The X loop starts at 0, counting up to XC-1 (one less than the number of characters needed on the X axis, since it started at zero).
460	SL, the current location in the string data, is incremented. This incremented number is used in line 465 as the memory location that the value from the video memory is POKEd into.

465 The next location in the video memory is calculated using the X and Y coordinates, and the character stored in that part of the video memory is POKEd into the string variable at location SL.
470 The X loop at the end of each horizontal line is closed up.
475 The SL variable is incremented again. If it is beyond the count required to contain the entire graphic, the program goes to the finish-up loop.
480 The linefeed character (26) is POKEd into the next character position at the end of the line, so the next character will begin on the following line on the screen.
485 At the end of each horizontal line, after character 26 effectively causes the cursor to go down one line, the Z loop begins. This loop (lines 485 to 500) is as long as the horizontal line (XC), and it POKEs character 8 (the BACKSPACE character) into each succeeding space. The effect is to backspace to the beginning of the next line on the video screen before starting the next row of characters.
490 The string location variable SL is incremented for this BACKSPACE loop.
495 The BACKSPACE character (8) is POKEd into the memory location.
500 The Z loop is closed up.
505 The Y loop is closed up, marking the end of the vertical lines required for the graphic.
510 A loop starts that will fill the rest of the variable with character 32 (SPACE). In this line, the string location variable is incremented. If it has reached the length of the string variable in memory (A$), this entire subroutine is done.
520 The SPACE character is POKEd into SL.
525 Here the subroutine ends.

USING THE SCREEN-EDITOR TO CREATE AN ARRAY OF GRAPHIC STRINGS

If you are using a TRS-80 Model III or Model 4, you can easily create an entire string array of graphics you create with this screen editor. (Model I users are not so fortunate: the following technique does not work on Model I's.)

Perhaps you want to create a group of graphic numerals to display large numbers on the screen. You may recall the program in chapter 4 that generated math problems and displayed them in large letters on the screen. To produce each large number, we used DATA statements that contained the numbers corresponding to each graphics character used in building the number.

Let's look at an alternate method using the program in Listing 6-H. Assume we want to use the string array N$() to contain the ten digits, with N$(0) as 0, N$(1) as 1, etc. The first step is to dimension our array. (Even though the default array size is 10, it is good practice to dimension it anyway. This makes it easier to change in case we want to add more elements to the array later.) We might use this line:

102 DIM N$(9)

Next we run the program and create our first large graphic character, the large number zero. After the string is drawn on the screen, press the "Y" key to begin building the string variable. In a few seconds the screen will return to its original status, with the X and Y counters back at 0,0. When this happens, press the <BREAK> key.

Line 110 will contain the graphic, and it will look something like this:

110 A$="
"
'(That's 128 spaces)'

(Your graphic should appear immediately after the first quotation mark.)

To edit the line, type

EDIT 110 <ENTER>

We will change A$ to N$(0). First, delete the A by typing

D

Insert the N$(0) by typing

I N $ (0)

Get out of the insert submode by holding

<SHIFT>-<UP ARROW>

(that is, hold the shift, and press the up-arrow key).

Hold the space bar down until the cursor is just past the <u>last</u> character in your graphic. If there are more blank spaces in the string, type

H " <SHIFT>-<UP ARROW>

Get out of the edit mode by pressing

<ENTER>

At this point, line 110 should look something like this:

110 N$(0)="   "

'(That's 128 spaces)'

(with the graphic after the first quotation mark).

Line 110 now assigns your graphic of the zero to N$(0). The next step is to add a new line which assigns the spaces to A$, and repeat the process.

To add the new line, type

111 A$="   "

'(That's 128 spaces)'

Repeat the process from the beginning, drawing the numeral 1 this time. Each time you finish, BREAK out of the drawing program, edit the line to change the variable name and chop off the end of the string, and add a new line that assigns 128 spaces to A$.

After all the graphics have been assigned, you may delete all the program lines except the string variable assignment statements, and write the rest of your program around them.

What are the advantages of using these packed variable assignment statements over READing numbers in DATA statements?

1) This program executes quickly. Quite a few string variables can be assigned in a second or so, whereas it took several seconds to assign eleven variables using the READ - DATA instructions.

2) The graphics take less program space. Each character on the screen created using this last method takes about 2 bytes of memory (1 for the character and one for the BACKSPACE under it). Each character on the screen took at least 4 characters using the READ - DATA method (three digits and a comma), and the extra programming to READ the DATA was required.

Many professional programmers use this methods for fast, space-efficient graphics in BASIC.

Chapter Summary

The VARPTR instruction, when used with a string variable, returns a value that points to the variable's location in memory.

The number stored at the VARPTR location is the length of the string pointed to.

The next two numbers are the least significant byte and the most

significant byte of the location where the string's data is stored.

The PEEK instruction can be used to determine the characters stored in the string.

The POKE instruction can be used to change the characters stored in the string.

By adding special control characters such as 8 (backspace) and 26 (linefeed), the graphics can be made to cover more than one line on the screen.

When string variable assignments are changed this way, the changes appear in the listing of the program, on the line where the string was originally assigned its value.

On the TRS-80 Model III and Model 4, the characters appear in the program listing just as they appear on the screen when the program is run. These graphics can be edited in BASIC's edit mode.

On the Model I, the characters appear as graphics when the program is run, but as reserved words when the program is listed. These graphics cannot be edited in BASIC's edit mode.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Write the description of the VARPTR instruction (as used with string variables).

B. For the VARPTR of a string variable, tell what is defined by each of the three bytes starting at the VARPTR location.

C. Given the three bytes starting at the VARPTR location, determine the memory location of the characters of the string variable.

D. Write a program that allows the user to view and edit the contents of the memory locations containing the characters of a string stored in a variable.

More Programming Practice

1. Write a program that contains an assignment statement that assigns a value to a string variable. It should then use the VARPTR instruction to determine the location of the string in memory. For each character in the string, it should then PRINT the ASCII value of the character and the character itself. The program should print this information horizontally, four characters per horizontal line.

2. Write a program that generates 15 random numbers between 128 and 191, then POKEs the value into the location of string data in

memory. It should print that string, pause, and generate another string.

3. Write a program that prompts the user for the length of a string and a graphics character number (128 to 191). It should then use the VARPTR instruction to locate a string variable's location in memory and POKE the appropriate characters into that area to create a vertical line of characters on the screen (with character 26 and character 8 between each adjacent pair). The line should be of the specified length and contain the specified character.

4. Write a program that prompts the user for a width, a length, and a graphics character number (128 to 191). It should then POKE the character value into the string storage area to create a string with the specified length and width, appearing to consist entirely of the specified character.

5. Using characters created by the program in Listing 6-H, write a program that PRINTs your name on the video screen.

Chapter Checkup

(For questions 1 to 3, assume the VARPTR of B$ is 31005.)

1. Write a BASIC program line that PRINTs the length of the string B$, without using the **LEN** instruction.

2. Write a BASIC program line that PRINTs the location of the first character of data in the string B$.

3. Write a BASIC program line that PRINTs the first character of the data in the string B$.

4. In line 30 of the program in Listing 6-G, why compare the value of VA to 2 15? If VA is greater than 2 15, why subtract 2 16?

5. In the program in Listing 6-H, why does the PRINT instruction on line 210 include character 30?

6. In the program in Listing 6-H, why does line 320 check the value of XM to be sure it is less than 127?

7. In the program in Listing 6-H, why does line 420 contain a **PRINT @ 768, " "** after the error message?

8. In the process of creating an array of string variables using the program in Listing 6-H, a new statement assigning spaces to A$ is needed for each new graphic. Why is it necessary to use A$ in these new lines, rather than some other string variable?

9. What is one way in which the VARPTR instruction is different from

any other BASIC instruction we have considered so far?

10. What is the maximum length of a string variable in an assignment statement in BASIC? (If you are not sure, see how long the longest one you can type in is.)

Chapter 7.
Using DEBUG under TRSDOS
To Pack String Graphics

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write the steps to be followed to view a program on disk in ASCII format using DEBUG under TRSDOS.
B. Use DEBUG to modify a program so that it assigns a string of character 191's (solid graphics characters) to a string variable.
C. Use DEBUG to modify a program so that it assigns the characters needed to form a rectangle 8 pixels wide and 9 pixels high to a string variable.
D. Use DEBUG to modify a program with a PRINT instruction, so that the program PRINTs a string of character 140.
E. Use DEBUG to combine adjacent lines of a BASIC program.
F. Use DEBUG to change the line numbers of a program to 0, thereby making the program difficult to alter.

INTRODUCTION

In the last chapter we looked at the VARPTR instruction which allows you to examine and alter the values in the memory locations which contain the data in string variables. By POKEing values for graphics characters directly into these locations, you changed the assignment statements within the programs to contain graphics data. This chapter discusses another technique for accomplishing the same thing and more by changing the program as it resides on a diskette.

Note: The techniques in this chapter work only with disk-equipped the TRS-80 Model III and the Model 4 (under Model III TRSDOS). They do not work on non-disk TRS-80 computers or under Model I TRSDOS.

EXAMINING THE PROGRAM UNDER DEBUG

To change a string variable assignment statement in a BASIC program so that it contains graphics characters, we must first examine the program under the TRSDOS utility DEBUG. Here are the steps to follow in this process, as well as an example program:

1. In Disk BASIC, write a program containing an instruction which assigns a value to a string variable. For example:

```
10 A$="          " '10 spaces
20 PRINT "A$=" A$
30 PRINT "LEN(A$)=" LEN(A$)
```

Listing 7-A

The assignment statement in this program is line 10 which assigns ten spaces to the string variable A$.

```
>LIST
10 A$="          " '10 spaces
20 PRINT "A$=" A$
30 PRINT "LEN(A$)=" LEN(A$)
READY
>RUN
A$=
LEN(A$)= 10
READY
>
```

Photo 7-A-1

2. SAVE the program on the diskette in the ASCII format. A bit of explanation is in order here.

When BASIC stores a program in the computer's memory or on a disk, it doesn't save every character you typed. Instead, it compresses the BASIC program by using special numbers called "tokens" in place of the reserved words. (Reserved words are the BASIC words that you use for programming: PRINT, IF, THEN, GOTO, etc.)

If you tell the computer to SAVE the program in the ASCII format, it SAVEs the program character by character as you typed it in. Since we are going to look at the program on the disk, we will SAVE it that way for easier viewing. To save the program in the ASCII format, type

SAVE "DEMO",A

and press ENTER. The A tells the computer to use the ASCII format.

3. Leave BASIC and go back to the operating system. To do this, type

CMD "S"

and press ENTER.

4. Get into DEBUG and look at the file. DEBUG is a TRSDOS function that allows us to look directly at the computer's memory and disk files. To get into DEBUG, when the **TRSDOS Ready** prompt appears, type

DEBUG

and press ENTER. Next the screen will show a lot of numbers and letters which represent part of the computer's memory. Type

F

and the **Filespec:** prompt will appear. Type in the filename you used to save the program. In this example, we will type

DEMO

and press ENTER. Another screen of numbers and letters will appear, this time representing the first portion of the file whose name we entered. This screen has three main parts:

+ The column of numbers on the left side of the screen tell us the drive number and record number of the part of the file we are examining.

+ The letters and numbers in the middle eight columns are the hexadecimal representations of the file contents (more on this shortly).

+ The letters and numbers on the right side of the screen are the ASCII representation of the file. This part shows the program as we typed it in.

Before proceeding, let's look at how these last two groups of numbers relate to each other. The first row of hex (that's short for hexadecimal) numbers reads

3130 2041 243D 2220 2020 2020 2020 2020

The first row of the the ASCII characters reads

10 A$="

followed by nine spaces.

The first two characters in the hex numbers (31) stand for the first

```
100100: 3130 2041 243D 2220 2020 2020 2020 2020 10 A$="
100110: 2022 2027 3130 2073 7061 6365 730D 3230  " '10 spaces 20
100120: 2050 5249 4E54 2022 4124 3D22 2041 240D  PRINT "A$=" A$
100130: 3330 2050 5249 4E54 2022 4C45 4E28 4124 30 PRINT "LEN(A$
100140: 293D 2220 4C45 4E28 4124 290D 6574 6169 )=" LEN(A$).etai
100150: 6C73 2C20 7365 6520 7061 6765 7320 3131 ls, see pages 11
100160: 3020 746F 2031 3131 2E22 00ED 6E28 0080 0 to 111.".5n(.
100170: 0009 6F32 00AD 224C 3649 3A31 223A 8522 ..o2. "L6I:1":"
100180: 443A 3122 3A89 4124 3A8D 3530 0000 003A D:1": A$: 50...
100190: 8522 443A 3122 3A89 4124 3A8D 3630 3000  "D:1": A$: 600.
1001A0: 0000 1C33 3630 20CD 2058 20CD 2059 CF36 ...360   X   Y 6
1001B0: 3429 0039 77D6 0120 2087 2058 0068 77DB 4).9w .   X.hw
1001C0: 0120 2053 4CD5 534C CD31 3A20 8F20 534C .  SL SL 1:   SL
1001D0: CE53 54D4 D520 434F 554E 5420 CA20 B120  ST   COUNT
1001E0: 534C 2C33 323A 8D20 3531 3000 7777 E001 SL,32:  510.ww
1001F0: 2020 B120 534C 2C20 3236 008C 77E5 0120    SL, 26.w
```

Photo 7-A-2

of the ASCII characters (1). The next two characters in the hex numbers (30) stand for the next ASCII character (0). Each pair of hex characters stands for the corresponding ASCII character (see Appendices 2 and 3).

The first row of hex characters ends with nine sets of the character pair "20". These stand for the nine spaces at the end of the ASCII side. These spaces (and the one on the next row) are the characters we will change to graphics characters.

5. Modify the contents of the data assigned to the string variable. To get into the modify mode, press

M

A cursor block will appear over the first character of the hexadecimal numbers in the middle columns of the screen. You can use the arrow keys to move the cursor block anywhere within the hexadecimal numbers.

Move the cursor to a position over the 2 in the first **20** in the group of nine 20's.

BFBFBFBFBFBFBFBFBFBF

and notice that the characters shown as "20" are changed to "BF". Also, the spaces shown on the ASCII portion on the right side have been changed to periods. (Only regular text characters are shown on the ASCII side; each of the others is represented by a period.) The cursor still appears on the screen to tell you that the characters have been changed on the display but not yet on the diskette.

Photo 7-A-3

After making changes in the modify mode, we may press ENTER to record the changes on disk or BREAK to cancel the changes typed in.

Press ENTER to record to the disk file the changes you have made.

6. Return to BASIC and load the program back into the computer's memory. Here are the steps:

+ First press BREAK to return to the "Filespec:" prompt.

+ Press BREAK one more time to get to the "TRSDOS Ready" prompt.

+ Type
BASIC <ENTER>
to load Disk BASIC back in.

+In response to the "Memory Size?" and "Number of Files?" prompts, press <ENTER>.

+Load in the altered program from disk by typing
LOAD "DEMO" <ENTER>.

+LIST out the program to see the changes. The program you see contains a string of character 191 (hex BF) in the assignment statement in line 10.

RUN the program to confirm that the string of graphic characters is assigned to A$.

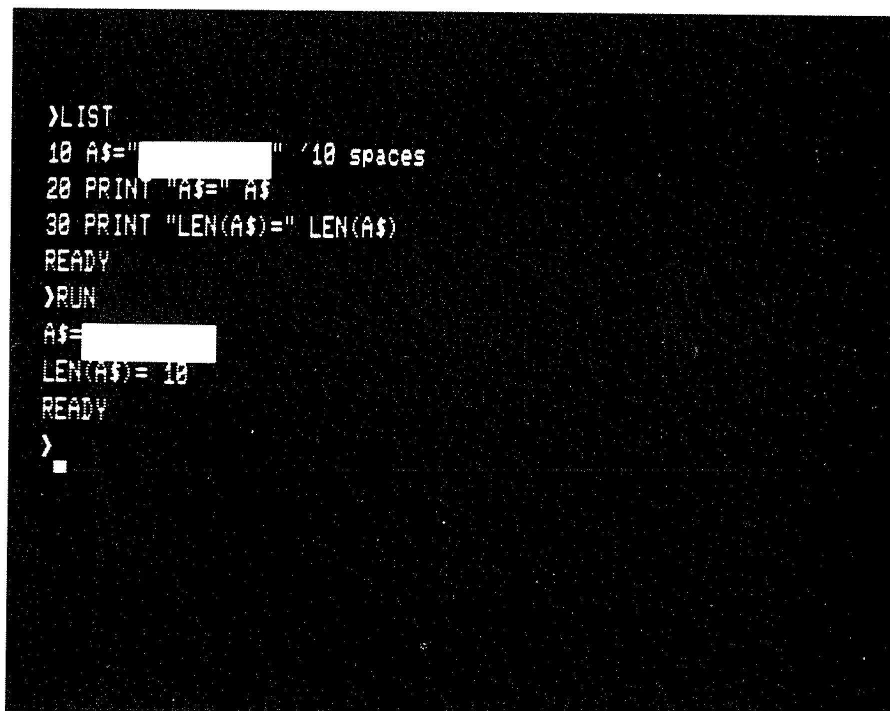


Photo 7-A-4

Using DEBUG Compared to Using the VARPTR

As you can see from the example, using DEBUG to pack string variables achieves the same results as using the VARPTR (as discussed in Chapter 6). For the programmer, here are some **advantages** of using DEBUG:

A. No special programming has to be added to the program when using DEBUG. When we used VARPTR, a special subroutine had to be added to the program to access the variables in memory. For some programs and some programmers, this may speed up the string-packing process considerably.

B. In DEBUG, you can always see the ASCII and hex values of the parts of the program being changed, so you are less likely to accidentally change the wrong part of the program.

C. DEBUG can be used to change strings in PRINT instructions as well as in assignment statements, as will be seen later in this chapter.

Here are some **disadvantages** of using DEBUG:

A. Only TRS-80 Model III and Model 4 disk-equipped computers can use DEBUG to alter strings. For non-disk users, this is a distinct disadvantage of DEBUG.

B. Changes made with DEBUG are made right on the disk, and changes made with VARPTR are made in the computer's memory. This makes changes performed with DEBUG more permanent, possibly making mistakes more serious.

C. Using the VARPTR, numbers typed in by the user may be either decimal or hex, depending on the BASIC program written. All numbers typed in under DEBUG must be in hex.

D. BASIC programs can be written to allow the user to see the string at all times. In DEBUG you cannot see any graphics characters on the screen.

ON WITH THE SHOW

Let's look at another example of the use of DEBUG to pack string variables.

Go back into DEBUG from "TRSDOS Ready" and look at the file DEMO once more. You should see the line of "BF" strings you typed in earlier. Position the cursor over the first "BF" string and type

BF BF BF 1A 08 08 08 BF BF BF

(Don't type in the spaces -- those are for ease of reading.)

The "BF" characters are the same graphics characters used before. "1A" in hex is the same as decimal 26, the move-down-one-line character. The "08" is our friend the backspace. So this string contains the following:

+ 3 graphics characters;
+ 1 linefeed to go to the next line;
+ 3 backspace characters; and
+ 3 more graphics characters.

The result is two sets of three graphics characters, one atop the other. When you go back to BASIC and look at the program, you will see the graphic in the first line has changed as described. In this example we see that **control characters can be added using DEBUG to create graphics covering more than one line on the screen.**

CREATING MORE COMPLEX GRAPHICS

We can create elaborate graphics on the video display and PEEK the characters used to create them from the video screen using the program in Listings 3-P, 3-Q, and 3-R. This program allows you to use the arrow keys to create the graphic. Then it PEEKs the value of each location in the video memory and PRINTs the locations and values of those that contain graphics. If you did Exercise 5 in the Chapter Checkup for Chapter 3, you have a version that LPRINTs these values on a printer instead of the video.

The author used this program to draw a diagonal line from the upper left corner of the display, approximately ten pixels long toward the lower right corner. While RUNning the screen-PEEKing routine, the program LPRINTed these values:

```
15360 137    15361 144        (row 1 on the screen)
15425 130    15426 164        (row 2 on the screen)
15491 137    15492 144        (row 3 on the screen)
15556 130    15557 132        (row 4 on the screen)
```

Figure 7-1

Positioned on the screen, these locations appear like this:

```
15360 15361
      15425  15426
                   15491  15492
                          15556  15557
```

Figure 7-2

As we can see from these characters' positions, control characters will be needed in the string we will build to position the graphics characters correctly.

Next let's look at the values to be typed into the string. Since our program prints out the values in decimal and DEBUG requires hex values, we need a way to convert from decimal to hex. Computers should work and people should work, so use the following program to do the conversion:

```
10 INPUT "DECIMAL NUMBER"; D
20 M = INT(D/16): L = D-M*16
30 IF M<10
   THEN M$ = RIGHT$(STR$(M),1)
   ELSE M$ = CHR$(M+55)
40 IF L<10
   THEN L$ = RIGHT$(STR$(L),1)
   ELSE L$ = CHR$(L+55)
50 LPRINT D "=" M$;L$
60 GOTO 10
```

Listing 7-B

(If you don't have a printer, use PRINT instead of LPRINT in line 50.)

Using this program, you type in decimal numbers and it LPRINTs the hexadecimal equivalents. RUN this program and type in the values PEEKed from the video memory (see Figure 7-1). The result is as follows:

137=89	144=90	130=82	164=A4
137=89	144=90	130=82	132=84

Figure 7-3

Now we know the relative positions of the locations (from Figure 7-2) and the hex values to type in (from Figure 7-3). We are ready to use DEBUG and put them into a program.

We have not calculated how many characters will be needed to build this string, so this program has some extras at the end of the string in line 10, just in case.

```
10 B$="                    " '20 spaces
20 PRINT B$
30 PRINT LEN(B$)
```

Listing 7-C

Save it under the filename "DEMO1" by typing

SAVE "DEMO1",A

and pressing ENTER.

Now return to TRSDOS, go into DEBUG, and examine the file "DEMO1".

The first two rows of characters show the ASCII representation of part of the program like this:

```
3130 2042 243D 2220 2020 2020 2020 2020
2020 2020 2020 2020 2020 2022 2027 3230
```

followed by the ASCII representation.

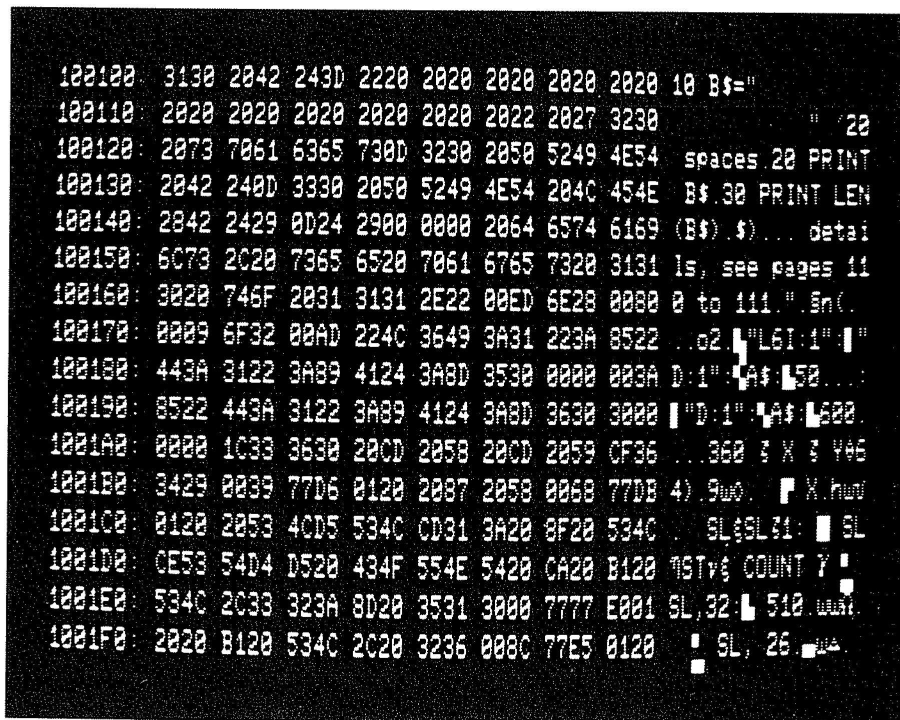


Photo 7-C-1

Press

M

to get into the modify mode. Position the cursor over the first 20 (space) after the 22 (quotation mark) and type in the values for the first two characters from Figure 7-3:

89 90

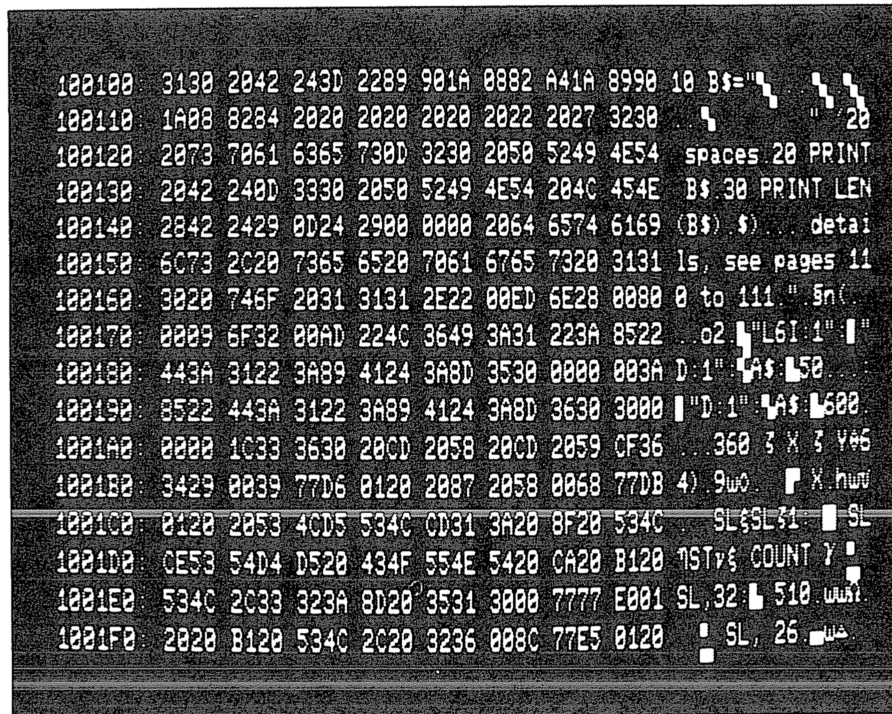


Photo 7-C-2

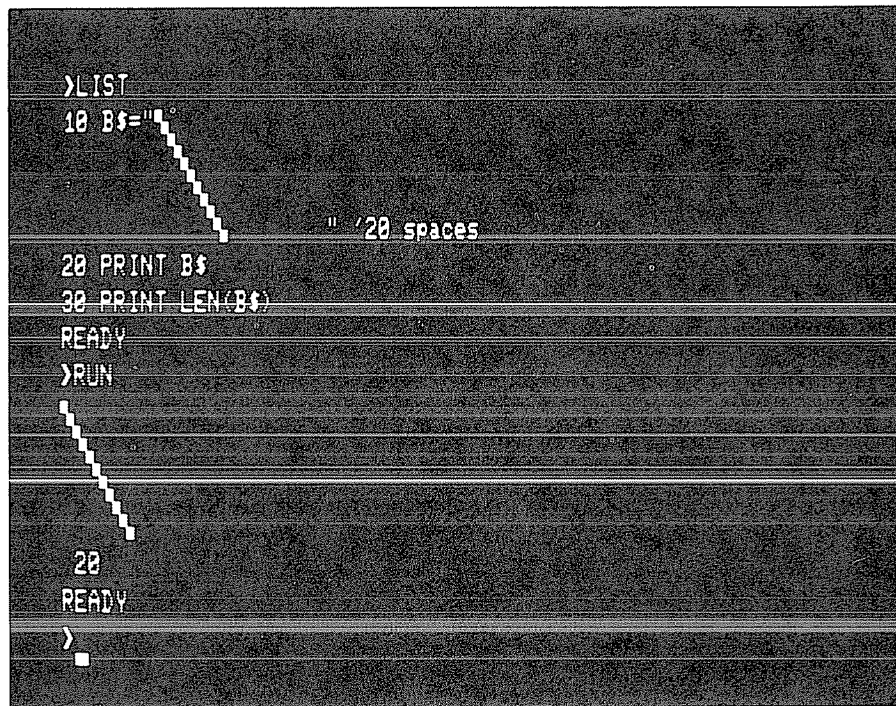


Photo 7-C-3

By referring to the positional chart in Figure 7-2, we see that the third position (15425) is directly under the second position (15361). To position the third character directly under the second one, we need to add characters to move down one position and back one position. type

1A 08

Type the next two characters:

82 A4

Referring again to Figure 7-2, we see the fifth position (15491) is down one and right one from the fourth position. Just one downward movement is needed, so type

1A

Type the next two characters:

82 84

Referring one last time to Figure 7-2, we see that the seventh position (15556) is directly under the sixth (15492). So type these characters to position it:

1A 08

Now type the last two characters:

82 84

Press **ENTER** to record the changes to the disk, then press **BREAK** two times to return the TRSDOS Ready.

Go back into BASIC and load **DEMO1** back in. Now when you run it, you will see the diagonal line just as it was originally drawn on the screen.

Further, you can edit line 10 of the program in BASIC's edit mode to remove the extra spaces from the end of the graphic string. (This makes the storage of the string more compact, and it may prevent problems with adjacent PRINTing of the string.)

Using DEBUG to Modify PRINT Instructions

As mentioned above in the advantages of using DEBUG over using the VARPTR instruction, you can modify a string of characters following a PRINT instruction in a BASIC program. This allows you to PRINT graphics characters efficiently without specifying their ASCII values in the program. To see how much memory this technique can save, let's look at an example.

To PRINT a string of graphics characters in an up-and-down form, we might use this sequence of characters: 141, 142, 131, 141, 142, 131, 141, 142, 131, 141, 142, 131, 141, 142. Putting these characters with a PRINT instruction, we have:

```
10 PRINT CHR$(141)+CHR$(142)+CHR$(131)+
         CHR$(141)+CHR$(142)+CHR$(131)+
         CHR$(141)+CHR$(142)+CHR$(131)+
         CHR$(141)+CHR$(142)+CHR$(131)+
         CHR$(142)
```

Listing 7-D

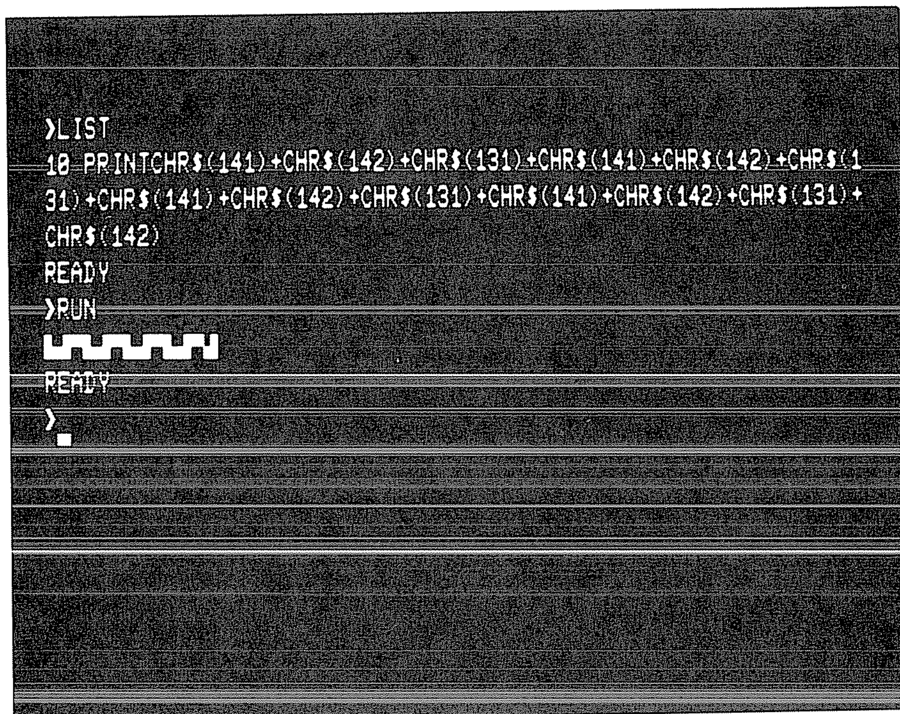


Photo 7-D

RUN this program to see that it displays the desired characters. On the author's Model III, the amount of memory left was 38096 bytes after CLEARing 50 bytes.

How can this be made more efficient? We can change this program to PRINT a string in quotation marks, then alter the program using DEBUG. Since the program in Listing 7-D contains fourteen graphics characters, let's use a program that PRINTs fourteen spaces, to begin with. Type in this program:

```
10 PRINT "              "
```

Listing 7-E

SAVE it in the ASCII format by typing:

SAVE "DEMO3",A <ENTER>

When you go into DEBUG and examine the file, you will find that the first two rows look like this:

```
3130 2050 5249 4E54 2022 2020 2020 2020 10 PRINT "
2020 2020 2020 2020 220D FFFF FFFF FFFF          "......
```

Figure 7-4

Go into the modify mode and position the cursor over the beginning of the second 20 (the one following the 22). Type in the hex values of the characters used in Listing 7-D as follows:

```
8D 8E 83   8D 8E 83   8D 8E 83   8D 8E 83
8D 8E 83   8D 8E
```

The graphic representation of these characters appears on the right side of the screen as you type them in. Record these changes on the disk file, return to BASIC, and load in DEMO3 again.

When you LIST the program, you will see that the graphics characters are indeed contained within the quotation marks following the PRINT instruction. RUN the program and you will see the same display as the program in Listing 7-D.

CLEAR 50 bytes of memory and PRINT the amount of memory left. On the author's Model III, 38177 bytes of memory remained. That represents a savings of 81 bytes over the program in Listing 7-D. Another

memory savings results because the original program uses string space to build the string of displayed characters, but this altered version uses no string space.

USING DEBUG TO COMBINE PROGRAM LINES

DEBUG's file-modifying capability has other useful applications for the BASIC programmer. One of these is combining two or more program lines into one line.

Why would you want to do this? Sometimes programs become too big to run properly in the computer's memory. The you must then "squeeze" the program to reduce the amount of memory it requires. Since each line number takes up five bytes of memory, combining lines can save quite a bit of memory.

One way to combine two adjacent lines in a program is simply to add the contents of one BASIC line to the previous line. But this method can be quite slow if you have very many lines to combine. Also, you are prone to errors whenever lines must be retyped. DEBUG provides us a way to combine BASIC lines without retyping.

To demonstrate this technique, start with this program:

```
10 FOR J=128 TO 191
20 PRINT "CHARACTER";
30 PRINT J;
40 PRINT "=";
50 PRINT CHR$(J);
60 PRINT " ",
70 NEXT J
```

Listing 7-D

RUN the program to be sure you have typed it in correctly. (It should PRINT all characters from 128 to 191 on the screen.) We are going to check the amount of memory this program occupies. Before we do, let's CLEAR 50 bytes of string space. By doing this each time we check the amount of memory left, we are assured that the amount of string space will not affect the memory check. Type:

```
CLEAR 50
```

Then to check the amount of memory, type:

```
PRINT MEM
```

On the author's 48K disk TRS-80 Model III, this returned 38115 bytes of remaining memory.

Here are the steps we must take to combine these program lines

into one line:

1. SAVE the program in the ASCII format. We will use the name **DEMO2.** Type:

SAVE "DEMO2",A <ENTER>

2. Go into DEBUG and examine the file. Type:

CMD"S" <ENTER>

DEBUG <ENTER>

F

DEMO2 <ENTER>

The file will now be displayed in hex and ASCII format. The first few lines look like this:

```
3130 2046 4F52 204A 3D31 3238 2054 4F20   10 FOR J=128 TO
3139 310D 3230 2050 5249 4E54 2243 4841   191.20 PRINT"CHA
5241 4354 4552 223B 0D33 3020 5052 494E   RACTER";.30 PRIN
```

The characters shown in bold letters are **0D.** Each stands for a carriage return (character 13, or 0D hex) that was pressed at the end of a program line.

3. Change each carriage return and line number that follows it to character 32 (20 hex), the SPACE character. Use the arrow keys to position the cursor block over the first **0D,** which follows the **31** in the second row. Type:

20

To replace the line number characters **3230,** type:

2020

Replace the next **0D** and line numbers **(3330)** with the space character **(20)** also.

The first lines of the screen should now show this:

```
3130 2046 4F52 204A 3D31 3238 2054 4F20   10 FOR J=128 TO
3139 3120 2020 2050 5249 4E54 2243 4841   191    PRINT"CHA
5241 4354 4552 223B 2020 2020 5052 494E   RACTER";    PRIN
```

The ASCII representation of the lines on the right side of the screen has also been changed, with spaces replacing the both periods that represented the carriage returns and the line numbers.

Continue through the file, changing the carriage returns and following line numbers to spaces. Be sure to leave the first line number as 10 so the program can be loaded back into BASIC!

When you have finished, record these changes to the disk file by pressing:

<ENTER>

4. Go to BASIC, load the program back in, and examine it. To return to the "TRSDOS Ready" prompt, press:

<BREAK> <BREAK>

Load in BASIC by typing:

BASIC <ENTER>

Respond to the "How many files?" and "Memory size?" questions by pressing:

<ENTER> <ENTER>

LOAD in DEMO2 by typing:

LOAD "DEMO2" <ENTER>

LIST the program. As shown in Listing 7-E, the whole program has been combined into one program line. RUN the program. You should get a "Syntax Error in 10" message since all the instructions in the program have been run together without any colons between them.

```
10 FOR J=128 TO 191    PRINT"CHARACTER";    PRINT J;
   PRINT "=";    PRINT CHR$(J);    PRINT" ",    NEXTJ
```

Listing 7-E

To correct the syntax error, edit the line. Insert a colon after each complete instruction, and delete the spaces before the next instruction. After you make these changes, line 10 should look like this:

```
10 FOR J=128 TO 191:PRINT"CHARACTER";:PRINT J;:PRINT "=";:PRINT
      CHR$(J);:PRINT" ",:NEXTJ
```

Listing 7-F

RUNning this program shows that it has no syntax errors, and that it performs the same function as the seven lines in Listing 7-D. To see how much memory we have saved by combining these lines, type:

CLEAR 50 <ENTER>

PRINT MEM <ENTER>

The author's Model III showed 38121 remaining bytes. This represents a savings of 38139 - 38115, or 24 bytes. This may not seem like much savings for that involved process, but sometimes 24 bytes can be significant. The author has spent dozens of hours looking for "a few more" bytes to allow a program to run in 16K!

Using DEBUG to Protect Program Code

Another interesting application of DEBUG's file modifying powers is to protect the lines of a BASIC program so they cannot easily be EDITed. We will do this by changing most of the line numbers to 0!

Let's use this as our sample program:

(See the note following the listing before typing it in.)

```
10 FOR J=1 TO 10
20 IF INT (J/2) = J/2
     THEN GOSUB 99
     ELSE GOSUB 199
30 NEXT J
40 END
99 REM
100 PRINT J "IS EVEN"
110 RETURN
199 REM
200 PRINT J "IS ODD"
210 RETURN
```

Listing 7-G

(Note: Type this program in exactly as it appears in the listing, including the spaces. In line 20, before the **THEN** and before the **ELSE**, press the down-arrow key and then insert six spaces.)

Since we are working on a protection technique, let's consider how it will work to protect this program. The usual way to alter a BASIC program is to edit the lines. To edit line 200, for example, we would type **EDIT 200**. But if all the line numbers were 0, only the first line 0 could be edited.

A problem arises when the program contains a branch to a program line. If all the line numbers were 0, the only way to branch would be

GOTO 0, and that would just branch to the first program line.

The solution to this problem is **to change only those lines which are not branched to.** And if all the lines that are branched to contain only REMarks, no important lines can be edited!

Our plan, then, is to SAVE the program onto a disk, go into DEBUG, and change the appropriate line numbers in the program to 0. If the file is saved in the ASCII format (as our previous programs have been), BASIC will reinterpret it into BASIC's "tokens" when it is read back from the disk. When we go into DEBUG and change the program numbers to zero, only one line 0 will be retained in the program!

If the file is saved in BASIC's compressed format, it is not reinterpreted as it is read in, and the line numbers in memory are left the same as they were on the disk. Because of this, we will save the program in BASIC's regular compressed format.

After you have typed in the program, save the program in the compressed format by typing:

SAVE "DEMO4"

Then go into DEBUG and examine the disk file called DEMO4. The first nine lines look like this:

```
FF8C 6A0A 0081 204A D531 20BD 2031 3000 ..j... J.1 . 10.
BB6A 1400 8F20 D828 4AD0 3229 20D5 204A .j... .(J.2) . J
D032 0A20 2020 2020 20CA 2091 2039 390A .2.          . . 99.
2020 2020 2020 3A95 2091 2031 3939 00C3          :. . 199..
6A1E 0087 204A 00C9 6A28 0080 00CF 6A63 j... J..j(....jc
0093 00E1 6A64 00B2 204A 2022 4953 2045 ....jd.. J "IS E
5645 4E22 00E7 6A6E 0092 00ED 6AC7 0093 VEN"..jn....j...
00FE 6AC8 00B2 204A 2022 4953 204F 4444 ..j... J "IS ODD
2200 046B D200 9200 0000 0000 0000 0000 "..k............
```

Figure 7-5

Now THAT doesn't look much like a BASIC program! But to BASIC, that is exactly how our BASIC program looks. We need to change the line numbers, so we will concentrate on how to find them in this display.

The first six bytes can be ignored for our purposes. The next four bytes are 0A 00. These represent the line number 10. To arrive at this, convert each pair of characters to decimal.

0A (hex) = 10 (decimal)
00 (hex) = 0 (decimal)

```
100100  FF8C 6E0A 0081 204A D531 20BD 2031 3000
100110: BB6E 1400 8F20 D828 4AD0 3229 20D5 204A
100120: D032 0A20 2020 2020 20CA 2091 2039 390A
100130: 2020 2020 2020 3A95 2091 2031 3939 00C3
100140: 6E1E 0087 204A 00C9 6E28 0080 00CF 6E63
100150: 0093 00E1 6E64 00B2 204A 2022 4953 2045
100160: 5645 4E22 00E7 6E6E 0092 00ED 6EC7 0093
100170: 00FE 6EC8 00B2 204A 2022 4953 204F 4444
100180: 2200 046F D200 9200 0000 0000 0000 0000
100190: 0000 0000 0000 0000 0000 0000 0000 0000
1001A0: 0000 0000 0000 0000 0000 0000 0000 0000
1001B0: 0000 0000 0000 0000 0000 0000 0000 0000
1001C0: 0000 0000 0000 0000 0000 0000 0000 0000
1001D0: 0000 0000 0000 0000 0000 0000 0000 0000
1001E0: 0000 0000 0000 0000 0000 0000 0000 0000
1001F0: 0000 0000 0000 0000 0000 0000 0000 0000
```

Photo 7-G

Multiply the second decimal number by 256 and add it to the first number.

(0 x 256) + 10 = 10.

So that represents the line number 10. Since the program in Listing 7-G has no branches to line 10, we can change this line number to 0. **Go into the modify mode of DEBUG and change the 0A to 00.**

Now that we know how the line numbers are displayed, we can look for the next one more easily. Line 20 is next, and 20 decimal is equivalent to 14 hex. Therefore, we need to look for an occurrence of "1400". It appears in the second line, in the second set of characters. Since there are no branches to line 20, we can change this line number to line 0. **In the modify mode, change the 14 to 00.**

In the same manner, change all the line numbers except 99 and 199 (which are branched to) to 0. They are as follows:

BASIC Line	Display Line	Find	Change To
30	5	1E00	0000
40	5	2800	0000
100	6	6400	0000
110	7	6E00	0000
200	8	C800	0000
210	9	D200	0000

Figure 7-6

After you have made these changes, your display should look like this:

```
FF8C 6A00 0081 204A D531 20BD 2031 3000 ..j... J.1 . 10.
BB6A 0000 8F20 D828 4AD0 3229 20D5 204A .j... .(J.2) . J
D032 0A20 2020 2020 20CA 2091 2039 390A .2.         . . 99.
2020 2020 2020 3A95 2091 2031 3939 00C3           :. . 199..
6A00 0087 204A 00C9 6A00 0080 00CF 6A63 j... J..j.....jc
0093 00E1 6A00 00B2 204A 2022 4953 2045 ....j... J "IS E
5645 4E22 00E7 6A00 0092 00ED 6AC7 0093 VEN"..j.....j...
00FE 6A00 00B2 204A 2022 4953 204F 4444 ..j... J "IS ODD
2200 046B 0000 9200 0000 0000 0000 0000 "..k............
```

Figure 7-7

Press **ENTER** to store your changes to the disk file. Then return to BASIC and load DEMO4 back in. Did you think it would look like it does?

```
0 FOR J=1 TO 10
0 IF INT (J/2) = J/2
    THEN GOSUB 99
    ELSE GOSUB 199
0 NEXT J
0 END
99 REM
0 PRINT J "IS EVEN"
0 RETURN
199 REM
0 PRINT J "IS ODD"
0 RETURN
```

Listing 7-H

Now you can edit lines 99, 199, and the first line 0, but the rest cannot be edited unless the line numbers are changed back. Try it!

Chapter Summary

The techniques used in this chapter work on disk-equipped TRS-80 Model III's and 4's under Model III TRSDOS 1.3 (the current version).

The steps to changing a string variable assignment statement so that it assigns graphics to the variable follow:

1. In Disk BASIC, write a program containing an instruction which assigns a value to a string variable.
2. SAVE the program on the diskette in the ASCII format.
3. Leave BASIC and go back to the operating system.
4. Get into DEBUG and look at the file.
5. Modify the contents of the data assigned to the string variable.
6. Return to BASIC and load the program back into the computer's memory.

Control characters may be added to the graphics string built under DEBUG to create more complex graphics.

Using a similar technique, you can use DEBUG to change a string of text characters following a PRINT instruction to a string of graphics characters. This can lead to a significant savings of memory space needed to store the program.

DEBUG can be used for other purposes besides graphics:

Using DEBUG to modify a program file saved in the ASCII format, you can replace line numbers with spaces to combine BASIC program lines. This, too, can lead to a significant memory savings.

A degree of protection from program changes can be reached by modifying a file saved in BASIC's compressed format by changing line numbers in the program to 0.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Write the steps to be followed to view a program on disk in ASCII format using DEBUG under TRSDOS.
B. Use DEBUG to modify a program so that it assigns a string of character 191's (solid graphics characters) to a string variable.
C. Use DEBUG to modify a program so that it assigns the characters

needed to form a rectangle 8 pixels wide and 9 pixels high to a string variable.

D. Use DEBUG to modify a program with a PRINT instruction, so that the program PRINTs a string of character 140.

E. Use DEBUG to combine adjacent lines of a BASIC program.

F. Use DEBUG to change the line numbers of a program to 0, thereby making the program difficult to alter.

More Programming Practice

1. Using DEBUG to modify a program file on disk, write a program that contains a PRINT instruction which PRINTs your first name in graphics characters.

2. Using DEBUG to modify a program file on disk, write a program that assigns a graphic to a string variable and PRINTs it. The graphic should PRINT as a vertical line 3 pixels wide and 20 pixels long.

3. Using DEBUG to modify a program file on disk, write a program that PRINTs a graphic shaped like a rectangle with the word "Box" in text characters spelled out inside the box.

4. Write a program containing several lines, each with a PRINT instruction. Using DEBUG, combine the lines into one line, and replace the PRINT instructions with semicolons (except the first PRINT). This should result in one program line with one PRINT instruction.

5. Using DEBUG to modify a program stored in the compressed format, change all lines of the program (except the lines that are branched to) to line 10.

Chapter Checkup

1. Type in the BASIC program in Listing 7-G. Save it in ASCII format under one filename, and save it in compressed format under another filename. How many bytes are used to store it under each format?

2. What percent savings over the ASCII format does the compressed format represent in this example?

3. Write the hex equivalents of these decimal numbers:
   A. 128 B. 132 C. 164 D. 191.

4. Which of these are characteristics of BASIC program files saved under the ASCII format, and which are charactistics of files saved under the compressed format?

   A) A four-character code represents the program line number.
   B) Reserved words are spelled out.

C) The end of each line contains a "carriage return" character (character 13 decimal).
D) Longer reserved words require more memory.
E) The program is stored in the computer's memory in this format.

5. Considering the way line numbers are stored in the compressed format, what is the biggest line number you can use in BASIC? What is the smallest line number you can use?

6. Consider this program:

```
10 FOR K=128 TO 191
20 PRINT K,
30 PRINT CHR$(K),
40 NEXT K
```

Knowing what you do about BASIC's reinterpretation of BASIC lines under certain circumstances, what would probably happen if you went into DEBUG and changed line number 20 to 30 and line number 30 to line 20:
A) if the program is saved on disk using the ASCII format; and
B) if the program is saved on disk using the compressed format?

7. Consider a program that would PRINT three packed strings, each 63 bytes long. The first string would PRINT a line with the top row of pixels set, the second string would PRINT a line with the middle row of pixels set, and the third line would PRINT a line with the bottom row of pixels set. Which character numbers would you use to pack into these strings? Write the decimal numbers and their hex equivalents.

8. It is possible for a program to PRINT a string of characters that goes beyond the last character on the video display. For example,

```
PRINT @ 1020, "XXXXXXXXXX"
```

What happens in this situation?

9. In your mind, what are the two most important advantages of using DEBUG under TRSDOS to pack graphics into BASIC programs? What are the two most important disadvantages?

10. Character 191 is a graphics character, character 8 is the backspace, and character 32 is a space. What would appear on the screen if you packed a PRINT instruction with a string of the following characters?

191 191 191 191 8 8 8 8 32 32 32 32.

Chapter 8.
Introduction to
Assembly Language Graphics Programming

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Define assembly language and its relationship to machine language.
B. Convert a decimal number to hexadecimal form.
C. Convert a hexadecimal number to decimal form.
D. Describe the uses in a Z-80 assembly language program of the following instructions: ORG, LD, INC, CP, JR, RET, END, and LDIR.
E. Given a listing of a assembled assembly language program, write a BASIC program that POKEs the assembly language program into memory and calls from BASIC.

Introduction

All programs presented so far in this book have been written in the programming language called **BASIC.** This is the high-level language found in most microcomputers, including the TRS-80 Model I, III, and 4. It is an easy-to-use language that many programmers learn before learning any other languages.

BASIC is called an "interpretive" language because every BASIC instruction is interpreted into machine language as the BASIC program is run. BASIC itself is a program that interprets your BASIC program into the Z-80 processor's "native" language, machine language.

Machine language programs are usually written in a sort of language called "assembly language." Assembly language is a rather cryptic language which has such seemingly simple instructions such as LOAD, INCREMENT, COMPARE, JUMP, AND RETURN. Each of these instructions does a small job, but together they can accomplish powerful programming tasks that simply cannot be done in BASIC.

The programmer usually writes or edits the assembly language program with a software tool called an "Editor." Unlike BASIC programs, assembly language programs cannot be RUN (executed) as they are written. Rather, the assembly language program written with the Editor must be **assembled** into the native machine language. An Assembler program is the software tool used for assembling them into machine language.

Working in assembly language to write machine language programs has its advantages and disadvantages compared with writing BASIC programs. On the plus side, programs that run in machine language invariably execute MUCH faster than programs written in BASIC. We shall soon see examples of this. Another advangage of machine language programs is that they give the programmer much more power to control

the inner workings of the computer.

A disadvantage of working in assembly language is that it is usually more difficult than programming in BASIC. Loading, moving, incrementing, comparing, jumping, and returning are a bit more abstract than PRINTs, GOTOs, LETs, IF - THENs, and ENDs. Another disadvantage for first-time assembly language programmers, is that programming in assembly language usually requires the use of the special Editor - Assembler software, which must be purchased separately.

Before getting too far into the descriptions of assembly language and machine language, let's look at a sample program written in assembly language. (The author intends in this section to give the reader an overview of assembly, not an in-depth study. Therefore, explanations about assembly language itself will be brief, except when they pertain specifically to graphics programming.)

```
00100 ; SAMPLE PROGRAM
00110 ; IN ASSEMBLY LANGUAGE
00120 ;
00130          ORG     48000D          ;START POINT IN MEMORY
00140 START:   LD      HL,16000D       ;LOAD HL W/ VIDRAM LOC.
00150          LD      (HL), 191D      ;LOAD VIDRAM W/ 191
00160          RET                     ;RETURN
00170          END     START           ;END PROGRAM
```

Listing 8-A

Listing 8-A shows the program as it would be written by the Editor program. Here are some things to notice about this listing:

+ The numbers at the left side are the line numbers, similar in function to BASIC's line numbers. Most Editor programs have automatic line numbering, so having the numbers increment by 10 is very easy.

+ The comments (parts of the line the program won't execute) are preceded by semicolons (";"). Lines 100, 110, and 120 are strictly comment lines. The rest of the program lines have comments at the right side, following the assembly code.

+ The assembly language instructions follow the line numbers. These instructions are interpreted by the assembler.

Let's take a brief look at each instruction and what it does.

ORG 48000D tells the computer to start ("originate") the program at memory location 48000. The assembler will assemble and store the program so the code for the line immediately following this line, is stored at the specified memory location.

LD HL,16000D tells the computer to load the register called HL with the number 16000. (More on registers later.)

LD (HL), 191D tells the computer to load the memory location pointed to by HL (in this case, 16000) with the number 191. The parentheses around HL specify the memory location pointed to by HL, rather than the register HL itself. We could accomplish the same function as this line in BASIC by typing POKE 16000,191.

RET tells the computer to go back to whatever routine called it, much as RETURN ends a subroutine in BASIC. Generally, each assembly language program ends with a RET instruction.

END START tells the computer that this is the end of the program, and it tells it to begin the execution of the program at the line labelled "START."

The effect of this entire program, then, is the same as POKE 16000, 191 in BASIC. Since 16000 is within the video memory, executing this program causes character 191 to appear on the screen.

After the programmer finishes writing and editing the assembly language program, the Assembler is called to assemble (interpret and put together) this program into the TRS-80's native language, Z-80 code. To say this anothe way, the **source code** (written with the Editor) is assembled into the **object code** (the Z-80 code). Listing 8-B shows the object code for this program as it appears to the left of the source code when the program is assembled.

```
BB80            00130            ORG     48000D
BB80    21803E  00140   START:   LD      HL,16000D
BB83    36BF    00150            LD      (HL), 191D
BB85    C9      00160            RET
B880            00170            END     START
```

Listing 8-B

The first column of characters (starting with BB80) stand for the memory locations in which the object code will be stored when this program is loaded into memory as a machine language program. The next column (starting with 21803E) shows the actual object code, the values that will be loaded into those memory locations to be executed. Both kinds of numbers are shown in their hexadecimal representations.

After the program is assembled, it is stored on a tape or diskette. It can be loaded in later and executed as a machine language program.

To summarize, the main steps in writing and executing an assembly language program are to write the program with the Editor, assemble it with the Assembler, store it onto a disk or tape, load the program back in, and execute it.

Decimal and Hexadecimal Numbers

You may have noticed that the source code of the program above contained the numbers 48000D, 16000D, and 191D. The "D" after each number stands for "decimal." The Assembler automatically converted each number into its hexadecimal (or "hex") equivalent. 48000D became BB80 (see first memory location shown). 16000D became 3E80 (although it is shown in reverse order, as these numbers always are in the Editor Assembler and in memory). 191D became BF in hex.

Before we progress any further in our look at assembly language, let's look at how to convert back and forth between decimal and hex numbers.

Decimal numbers are the numbers we use every day. They are base 10 numbers. Since we have ten fingers, our number system is based on 10. From right to left, the numbers in a Base 10 number stand for the 1's, 10's, 100's, 1000's, etc. Each time the place value is multiplied by ten to get the next one.

The Base 10 system has ten different numerals, 0 through 9. When counting, each time you get to 9, you go on to 0 and add one to the next column to the left.

The hexadecimal number system is based on the number 16. It has sixteen "numerals," 0 to 9 and A to F. Instead of the place values of 1, 10, 10x10, 10x10x10, etc., the hex system has 1, 16, 16x16, 16x16x16, etc. Here is a chart showing the decimal and hex numbers with the same actual value:

Decimal	Hex	Decimal	Hex
1	1	17	11
2	2	18	12
3	3	19	13
4	4	20	14
5	5	21	15
6	6	22	16
7	7	23	17
8	8	24	18
9	9	25	19
10	A	26	1A
11	B	27	1B
12	C	28	1C
13	D	29	1D
14	E	30	1E
15	F	31	1F
16	10	32	20

Figure 8-1

Looking at these kinds of numbers in another way, let's compare the same string of digits as a hex and a decimal number to see the difference. The most common notation to specify decimal and hex numbers is a D and a H, respectively, as shown in Figure 8-2.

```
13579D  =
        1 x 10x10x10x10     +
        3 x 10x10x10        +
        5 x 10x10           +
        7 x 10              +
        9 x 1

13579H  =
        1 x 16x16x16x16     +
        3 x 16x16x16        +
        5 x 16x16           +
        7 x 16              +
        9 x 1

        =
        1 x 65536           +
        3 x 4096            +
        5 x 256             +
        7 x 16              +
        9 x 1

        =
        65536+12288+1280+112+9 = 79225D
```

Figure 8-2

The steps shown in Figure 8-2 can be used to convert any hex number to a decimal number. Converting a decimal number to a hex number requires reversing the process. Figure 8-3 shows the steps taken to convert 60000D to a hex number. After determining the largest power of 16 that is smaller than the decimal number, a series of divisions and multiplications, and subtractions is done. The largest power of 16 less than 60000D is 16x16x16 (which is 4096).

```
60000 / 4096 = 14.6484          2656 / 256 = 10.375
       4096x14= 57344                256x10=2560
60000 - 57344 = 2656            2656 - 2560 = 96

96 / 16 = 6                     0 / 1 = 0
   16x6=96                         1x0=0
96 - 96 = 0                     0 - 0 = 0
```

Using each main quotient, our digits in the hex number will be:

14 10 6 0

Converted to hex numbers, that's

E A 6 0.

So 60000D = EA60H.

When programming in assembly language, the programmer may specify either decimal numbers followed by a "D" or hex numbers followed by an "H" in the source code. The object code which the assembler generates will always contain the numbers in hex form.

A Word About Registers

The Z-80 microprocessor, the heart of the TRS-80 microcomputer, uses several registers to manipulate numbers. **Registers** can be thought of like memory locations, but they cannot be addressed like memory locations with POKEs and PEEKs. These eight-bit registers can be manipulated only in machine language, and they are what we use to manipulate the contents of other memory locations.

The Z-80 processor has the following pairs of registers: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY, the stack pointer, and the program counter. The ones used most by beginning programmers are AF, BC, DE, and HL. The program in Listing 8-A uses the HL register to store the video memory location into which the number 191D (or BFH) is loaded. The registers that can be used for the most data manipulations are the A register (the accumulator) and the HL register.

The F register (the counterpart of the A register) is the flag register. Its eight bits are used as flags, and they are automatically **set** (loaded with the number 1) or **reset** (loaded with a zero) according to the result of certain operations.

You will see the use of these registers in the sample programs that follow.

A Second Example: Filling the Screen

Here is a program which fills the screen with a character 191.

```
100 ;*********************************
110 ;       Screen Fill Program
120 ;         By Dennis Tanner
130 ;*********************************
140 ;
```

```
BF68            150          ORG  49000D      ;START POINT IN MEMORY
BF68  21FF3B  160  START: LD   HL,15359D  ;BEFORE BEG. OF VIDRAM
BF6B  23          170  LOOP:  INC  HL               ;NEXT SCREEN POSITION
BF6C  7C          180          LD   A,H              ;MOST SIG. BYTE VIDRAM
BB6D  FE40      190          CP   40H              ;BEYOND VIDRAM?
BF6F  C8          200          RET  Z                ;IF YES, RETURN
BF70  36BF      210          LD   (HL),191D    ;ELSE 191 TO VIDRAM
BF72  18F7      220          JR   LOOP            ;GO BACK TO "LOOP"
BF68              230          END  START          ;END PROGRAM
```

Listing 8-C

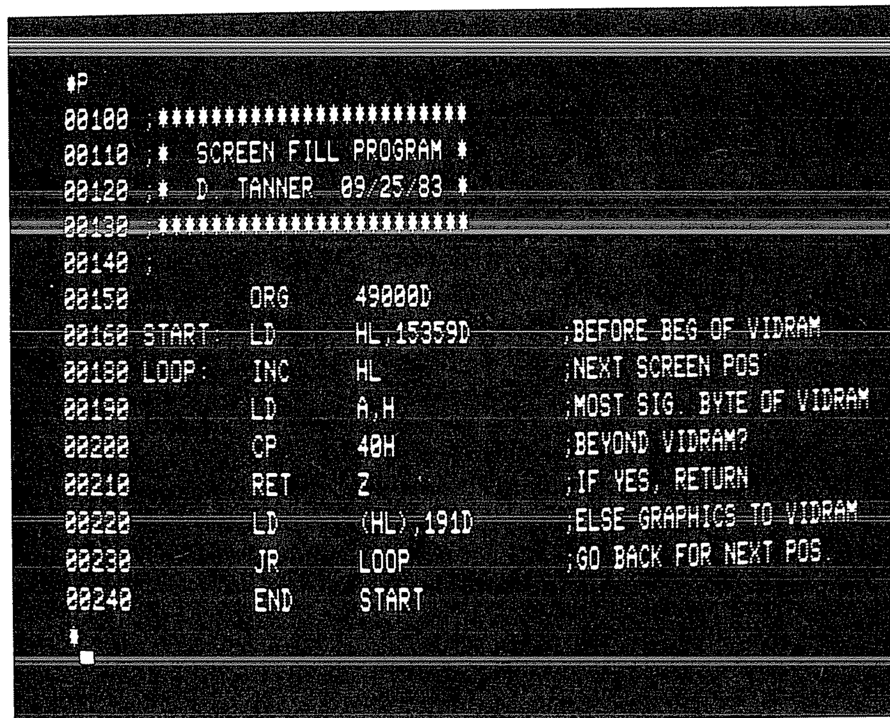


Photo 8-C

Here is a line-by-line description of this short program:

Line Description

100 to 140
Comment lines

150 Tells the assembler to assemble the program so that the first bytes of the program will load into memory location 49000D (that's BF68F).

160 Labelled **START**, this instruction loads the number 15359D (3BFFH) into the HL register. The video memory starts at the next memory

location (15360D or 3C00H). When HL is incremented in the next instruction, HL will contain the address of the beginning of video memory.

170 This line (labelled **LOOP**) increments (adds 1 to) the HL register pair. The effect is to point to the next location in the video memory.

180 This line loads the value in H to register A (the accumulator). Since HL first contains hex 3C00, the first two characters (3C) are contained in H. So the first time this is executed, 3C is loaded into the accumulator.

190 The CP instruction **COMPARES** the number on the program line following the instruction to the value stored in A. Now A contains 3CH, and it is compared with 40H.

200 This instruction tells the processor to RETurn if the Zero flag is set; that is, if the COMPARE instruction immediately above found an exact match. So when the first two characters of the video memory address stored in HL are 40, the program executes a RETurn. You may recall that the video memory runs from 15360D to 16383D, or 3C00H to 3FFFH. When you add one to 3FFFH, you get 4000H. So when the HL counter gets past the video memory, the program returns to whatever called it.

210 If the program didn't return, it continues to this line. This line loads a character 191D (BFH) to the location pointed to by the HL pointer. Remember that the parentheses around HL mean that the instruction will act not on HL itself, but rather on the address pointed to by HL. This is the instruction in this program that puts the character on the screen.

220 This instruction causes program control to go back to the address labelled **LOOP**, completing the program loop.

230 The assembler will stop assembling here.

Listing 8-D shows a BASIC program that does the same thing as the assembly language program in Listing 8-C.

```
10 'SLOW SCREEN FILLING PROGRAM
20 FOR J=15360 TO 16383
30 POKE J, 191
40 NEXT J
```

Listing 8-D

If you are working on a disk computer without the benefit of an Editor Assembler, you can still see the relative speed of the machine language program by POKEing the machine language program into memory and executing it. The following BASIC program shows you how.

```
10 'BASIC Screen Filler with machine language.
   Protect memory at 48000D
20 CLEAR 500: GOTO 110
30 FOR J=0 TO LEN(A$)/2-1
40  TE$= MID$(A$,J*2+1,2)
```

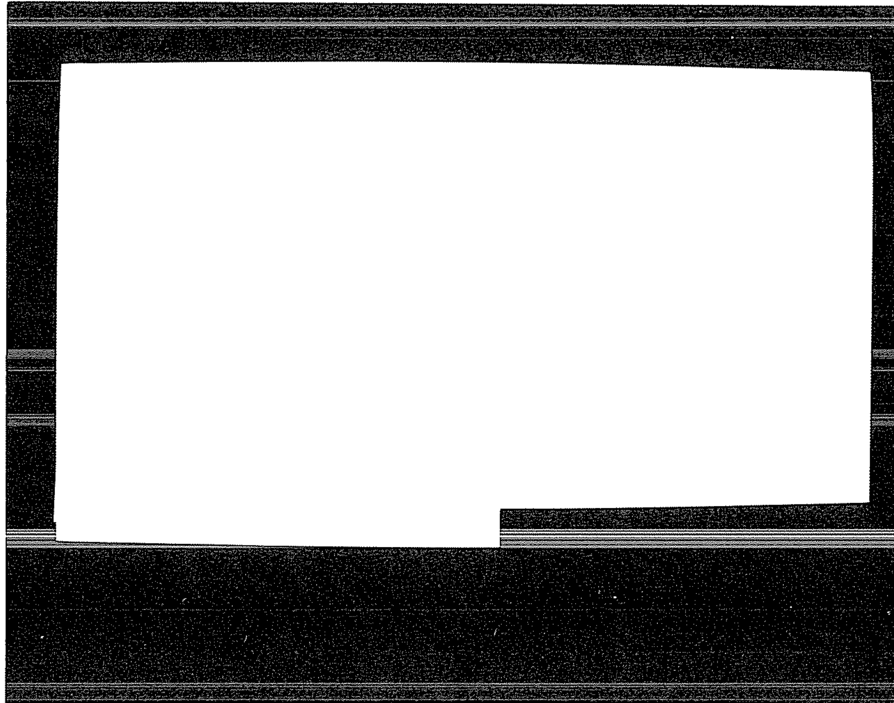


Photo 8-D

```
50  MSB= ASC(LEFT$(TE$,1)): MSB=MSB-48:
    IF MSB>9 THEN MSB= MSB-7
60  LSB= ASC(RIGHT$(TE$,1)): LSB=LSB-48:
    IF LSB>9 THEN LSB= LSB-7
70  NU= MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 A$= "21FF3B237CFE40C836BF18F7"
120 BG= 49000- 2 16
130 DEFUSR1= BG
140 GOSUB 30
150 X=USR1(0)
160 FOR J=1 TO 200:NEXTJ:CLS
170 FOR J=1 TO 200: NEXTJ: GOTO150
```

Listing 8-E

Lines 30 to 100 of this classy little program let you easily put a wide variety of assembly language programs into memory and run them, all from BASIC.

Line 10 reminds the user to protect memory at 48000 before running this program. Here's why this must be done. When run a program in BASIC, you usually don't care where in memory the program is stored. You need to know only what variables store what values. But in this program, the values which constitute the machine language are to be POKEd into a certain part of memory. If this part of memory is not protected before the program is run, BASIC might put variables, tables, or the BASIC program in the same memory area where the machine language program is POKEd, resulting in a nonfunctional program. So by protecting memory, you assure that BASIC will not infringe on the machine language program.

There are two common ways to protect memory at 48000: 1) type 48000 in response to the "Memory size?" question, or 2) enter Disk BASIC from TRSDOS in this format:

BASIC -M:48000 **ENTER**

Notice that line 20 of the program in Listing 8-E "jumps around" the subroutine to line 110. Lines 110, 120, and 130 are executed before the subroutine is called in line 140. Let's take a look at these lines to see what information must be passed to this subroutine.

Line 110 assigns a string of characters to the variable A$. **The characters assigned to A$ in this line represent the hexadecimal representation of the object code shown in Listing 8-C.** Look back to Listing 8-C, and you will see these same characters between the memory locations and the line numbers in the listing.

Line 120 assigns the memory location where the program will begin in memory to the variable BG.

Line 130 assigns the first USR call to BG (in this case, 49000D), the location where the machine language program execution will begin.

The subroutine is then called. It reads the string defined as A$ and analyzes it to determine the value of the hex numbers it contains. It then POKEs those values into the appropriate memory locations, starting at BG. Then the subroutine returns. When the BASIC program calls the machine language program with the USR call in line 150, control is transferred to the machine language program. It executes, filling the screen, then RETurns, transferring control back to the BASIC program.

If you have a disk TRS-80 Model III or Model 4, you should run this program. Be sure you understand what variables must be assigned before the BASIC subroutine is called. **This same subroutine will appear in each remaining chapter in this book as an easy way to execute machine language from BASIC.**

To demonstrate the speed of machine language, compare the time it takes to run the BASIC screen-filling program (Listing 8-D) to the time it

takes to run the machine language version (Listing 8-E). On the author's Model III, the BASIC routine took about 6 seconds, whereas the machine language version could execute at least 75 times in that same amount of time, with a CLS before each execution!

Moving Information Within the Computer

Sometimes a programmer wants to move information from one part of memory to another. Perhaps a block of characters makes up a person's name, and the programmer needs to move it from a volatile memory location (such as the video memory) to a relatively protected area (such as very high memory). Or the programmer may need to move a group of graphics characters in high memory to video memory. A simple BASIC program like that in Listing 8-F would do the job.

```
10 'Slow memory mover
20 CLS
30 INPUT"MOVE FROM";MF
40 INPUT"MOVE TO";MT
50 INPUT"HOW MANY";HM
60 FORJ=0 TO HM-1
70  POKE MT+J, PEEK(MF+J)
80 NEXT J
```

Listing 8-F

If you run this program and ask it to move data FROM 1000 (part of the computer's ROM) TO 15360 (the video memory) with a quantity of 1024 bytes, it will move data from the ROM onto the screen. It does take a while, though.

In machine language we can do this much faster. Z-80 assembly language has an instruction called **LDIR** which is one of the author's favorites. It moves data from one area of memory to another very quickly. Let's take a look at how the LDIR instruction might be used.

```
            100  ;LDIR Sample PROGRAM
            110  ;By Dennis Tanner
            120  ;
BB80        130        ORG  48000D    ;MEMORY START POINT
BB80 210000 140  START:LD   HL,0000   ;MOVE FROM LOCATION 0
BB83 11003C 150        LD   DE,3C00H  ;MOVE TO 15360D
BB86 010004 160        LD   BC,400H   ;COUNTER = 1024D
BB89 EDB0   170        LDIR           ;MOVE IT!
BB8B C9     180        RET            ;RETURN
BB80        190        END  START     ;END OF PROGRAM
```

Listing 8-G

The source code of the program (that which was typed in with the Editor) contains both decimal numbers (48000D) and hex numbers (3C00H

and 400H). The columns from left to right again represent memory locations, object code, line numbers, labels, source code (two columns), and comments.

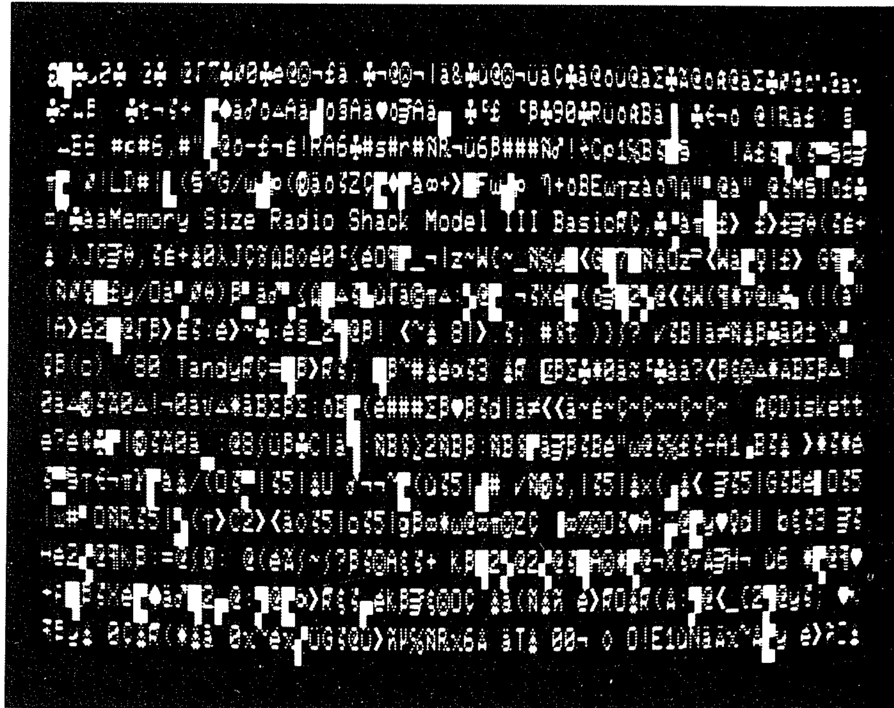


Photo 8-G

Here is a detailed description of the program.

Line	Description
100	Comments
110	Comments
120	Comments
130	Tells the assembler to assemble this program so that the first part of if will be stored at memory location 48000D (BB80H).
140	Loads the HL register pair with the value 0. **For the LDIR instruction, the HL pair always stores the "move from" location.**
150	Loads the DE register pair with the value 3C00H (15360D), the beginning of the video memory. **For the LDIR instruction, the DE pair always stores the "move to" location.**
160	Loads the BC register pair with 400H (1024D), the number of characters on the video screen. **For the LDIR instruction, the BC pair always stores the "counter" number.**
170	Here is the LDIR instruction. It performs the following steps: A) Moves the character or value stored in the location pointed to by HL (0) to the location pointed to by DE (15360D). B) Decrements (subtracts 1 from) BC, the counter.

C) If BC is not 0, go back to A.
D) If BC is 0, go to next Z-80 instruction.

180 Returns to whatever program called it.
190 Ends the program and tells the assembler to set the program execution point at the location labelled "START" (BB80).

Very quickly, these instructions move the contents of the beginning of ROM (from memory location 0 to location 1023) to the video memory, thus displaying them.

This is a good demonstration program, but it is admittedly of little practical value because of its single function. If a program allowed the user to select the "move from" and "move to" locations and the "how many" counter, it would be of much greater utility. The program in Listing 8-H does just that.

```
                100 ;
                110 ; Flexible LDIR program
                120 ;
BB80            130        ORG  48000D     ;MEMORY START POINT
BB80 2100BC     140 START: LD   HL,0BC00H  ;MSB OF "MOVE TO"
BB83 56         150        LD   D,(HL)     ;LOAD D WITH MSB
BB84 2101BC     160        LD   HL,0BC01H  ;LSB OF "MOVE TO"
BB87 5E         170        LD   E,(HL)     ;LOAD E WITH LSB
BB88 2102BC     180        LD   HL,0BC02H  ;MSB OF COUNTER
BB8B 46         190        LD   B,(HL)     ;LOAD B WITH MSB
BB8C 2103BC     200        LD   HL,0BC03H  ;LSB OF COUNTER
BB8F 4E         210        LD   C,(HL)     ;LOAD C WITH LSB
BB90 3A04BC     220        LD   A,(0BC04H) ;MSB OF "MOVE FROM"
BB93 67         230        LD   H,A        ;LOAD H WITH MSB
BB94 3A05BC     240        LD   A,(0BC05H) ;LSB OF "MOVE FROM"
BB97 6F         250        LD   L,A        ;LOAD A WITH LSB
BB98 EDB0       260        LDIR            ;MOVE IT!
BB9A C9         270        RET             ;RETURN
BB80            280        END  START      ;END OF PROGRAM
```

Listing 8-H

This program "PEEKs" the values for "move to", "how many," and "move from" out of specific memory locations. (The author arbitrarily chose BC00H and BC01H for "move to," BC02H and BC03H for "how many," and BC04H and BC05H for "move from.") Here is how this program works:

+ Lines 140 to 210 load the number representing the memory location holding each part of the number into the HL register pair. then Then the number pointed to by HL (and thus contained in the memory location) is loaded into the appropriate register.

+ Lines 220 to 250 load the number pointed to by the "move from" locations to the A register, then to the appropriate H or L register.

+ Line 260 contains the LDIR which executes the move.

So the numbers stored in memory locations BC00H to BC05H determine the "move from," "move to," and "how many" locations.

The next obvious question is how to get the numbers we want into those locations. Since BASIC can call this machine language routine, let's use a BASIC program as a means of sending the machine language program the appropriate values. The program in Listing 8-I shows how.

```
10 'LDIR/BAS  Flexible LDIR BASIC program
            Protect memory at 48000
20 CLEAR500: GOTO160
30 FOR J=0 TO LEN(A$)/2 -1
40  TE$ = MID$(A$,J*2+1,2)
50  MSB = ASC(LEFT$(TE$,1)): MSB=MSB-48:
            IF MSB>9 THEN MSB=MSB-7
60  LSB = ASC(RIGHT$(TE$,1)): LSB=LSB-48:
            IF LSB>9 THEN LSB=LSB-7
70  NU = MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 POKE &HBC00, INT(MT/256):
            POKE &HBC01, MT-INT(MT/256)*256
120 POKE&HBC02, INT(HM/256):
            POKE&HBC03, HM-INT(HM/256)*256
130 POKE&HBC04, INT(MF/256):
            POKE&HBC05, MF-INT(MF/256)*256
140 X = USR1(0)
150 RETURN
160 A$="2100BC562101BC5E2102BC462103BC4E3A04BC673A05BC6F
EDB0C9"
170 BG = 48000-2[16
180 DEFUSR1 = BG
190 GOSUB 30
200 MF = 5600: MT = 15360: HM = 1024: GOSUB 110
210 GOTO 210
220 END
```

Listing 8-I

This program calls the machine language program that appears in Listing 8-H. Its format is similar to that of Listing 8-E. It contains a BASIC subroutine which uses the variables MT (move to), MF (move from) and HM (how many). This subroutine then calls the machine language program, then RETURNs. Here is an analysis of the program:

Line Description

10 to 100
Same as that of Listing 8-I. This routine analyzes A$ (which contains the machine language routine in string form) and POKEs it into memory beginning at the location specified by BG.

110 to 130
This is the beginning of the subroutine that is called to execute the machine language program. These lines analyze the variables MT, HM, and MF, and POKE their values into the appropriate memory locations to be read by the machine language routine. Since machine language programs can deal with numbers only up to 255 (eight-bit numbers), the larger numbers in MT, HM, and MF must be broken down into smaller numbers. These lines find the **most significant byte** and **least significant byte** which are then POKEd into memory.

140 The user call in this line executes the LDIR machine language program that was POKEd into memory in lines 30 to 90.

150 RETURNs from the subroutine.

160 Line 20 jumps to this line. It assigns the ASCII form of the object code to the variable A$. (See the second column of Listing 8-H.)

170 BG is assigned the value of the memory location where the machine language program will begin. 2^{16} (65536) is subtracted from the memory location value (48000) because POKEs can be used only on numbers between -32768 and 32767.

180 The first user call (USR1) is defined at the same location, BG. This means that when the routine is called (line 140), program control will transfer to that location.

190 After A$ and BG are assigned their values, this line calls the subroutine which POKEs the machine code program into memory.

200 The MF (move from) value is set at 5600, an area of the ROM memory. The MT (move to) value is set at 15360, the beginning of the video memory. The HM (how many) value is set at 1024, the number of characters on the video display. Then the line calls the subroutine which in turn calls the machine language program.

210 This line loops back to itself until the BREAK key is pressed, holding the video display.

The area of the ROM memory displayed by this program shows the reserved words and two-character error messages as they are stored in the ROM. Would you like to see the rest of the computer's memory? Just make these changes to the program in Listing 8-I:

```
200 MF=0: MT=15370: HM=1014
210 CLS: PRINT @ 0, MF;: GOSUB 110: I$=INKEY$
220 I$=INKEY$: IF I$=CHR$(10)
                    THEN IF MF<1014 THEN 220
                    ELSE MF=MF-1014: GOTO 210
```

```
230 IF I$=CHR$(91)
      THEN IF MF>64522 THEN 220
      ELSE MF=MF+1014: GOTO210
240 GOTO 220
```

Listing 8-J

An explanation of this program is left to the reader. (See the Chapter Checkup.)

A View Inside the Computer

As the previous program shows, this LDIR command can be helpful in making parts of memory "visible" to the user by moving them to the video memory. The speed with which it makes the move gives this technique special application for the video memory.

Your TRS-80 microcomputer performs many functions you don't see as a BASIC programmer. One of these is the storage of strings. Do you remember the CLEAR instruction which sets aside a certain number of bytes of memory for BASIC to handle its strings? Have you ever wondered how and where those strings are stored?

The strings are stored just below the top of the memory usable by BASIC. If no memory is protected, the strings are stored near the top of memory. If memory is protected at 48000D, for example, the strings are stored just below that. By using the LDIR fast move routine to move the values from that area to the video memory, we can see what happens in that string storage space.

```
10 'VISSORT/BAS
   By Dennis Tanner
   PROTECT MEMORY AT 48000
20 CLS: CLEAR62: GOTO160
30 FOR J=0 TO LEN(A$)/2-1
40   TE$ = MID$(A$,J*2+1,2)
50   MSB = ASC(LEFT$(TE$,1)): MSB=MSB-48:
     IF MSB>9 THEN MSB= MSB-7
60   LSB = ASC(RIGHT$(TE$,1)): LSB=LSB-48:
     IF LSB>9 THEN LSB= LSB-7
70   NU = MSB*16+LSB
80   POKE BG+J, NU
90 NEXT J
100 RETURN
110 POKE &HBC00, INT(MT/256): POKE &HBC01, MT -INT(MT/256) *256
120 POKE &HBC02, INT(HM/256): POKE &HBC03, HM -INT(HM/256) *256
130 POKE &HBC04, INT(MF/256): POKE &HBC05, MF -INT(MF/256) *256
140 X = USR1(0)
150 RETURN
160  A$="2100BC562101BC5E2102BC462103BC4E3A04BC673A05BC6F
EDB0C9" 'LDIR/CMD
```

```
170 BG = 48000 - 2[16
180 DEFUSR1 = BG
190 GOSUB 30
200 PRINT @ 128, "The string space is shown above."
210 PRINT@320,"Strings to be sorted:"
220 FOR J=1 TO 10
230 A$(J) = STRING$(RND(5),J+64)
240 PRINT A$(J); " ";
250 NEXT J
260 FOR Z=1 TO 9
270   FOR Y=Z+1 TO 10
280     GOSUB370
290     IF A$(Z) < A$(Y)
        THEN B$=A$(Z): A$(Z)=A$(Y): A$(Y)=B$
300   NEXT Y
310 NEXT Z
320 PRINT @ 576, "Sorted strings:"
330 FOR J=1 TO 10
340  PRINT A$(J); " ";
350 NEXT J
```

Listing 8-K

Lines 20 through 190 are the same as the corresponding lines in Listing 8-I. Thus the same LDIR machine language routine is available. Let's look a little more closely at lines 200 to 370.

Lines 200 to 250 assign a string of random length and having ASCII value 65 to 74, to the A$() array. This would require 50 bytes of string space at most, and 62 bytes have been cleared (see line 20).

Lines 260 to 310 sort the strings. This is a bubble sort, an inefficient but easy-to-understand sort. Notice that once per loop, subroutine 370 is called.

Lines 320 to 360 display the sorted strings and end the program.

Line 370 is the subroutine called from line 270. It assigns 47937 to the "Move From" variable, 15360 to the "Move To" variable, and 60 to the "How Many" variable before calling the LDIR subroutine beginning in line 110. 47937 is just below 48000, the top of memory available to BASIC.

This is a fascinating program to watch. The strings assigned in line 230 are displayed first in the string space area (which is echoed at the top of the screen). As more strings are compared, assigned, and reassigned, BASIC gives them another part of the string space. When the string space appears to be full, BASIC pauses momentarily to move all the currently assigned strings to the top of memory, thus freeing up a little more string space. All this manipulation is visible during the execution of this program.

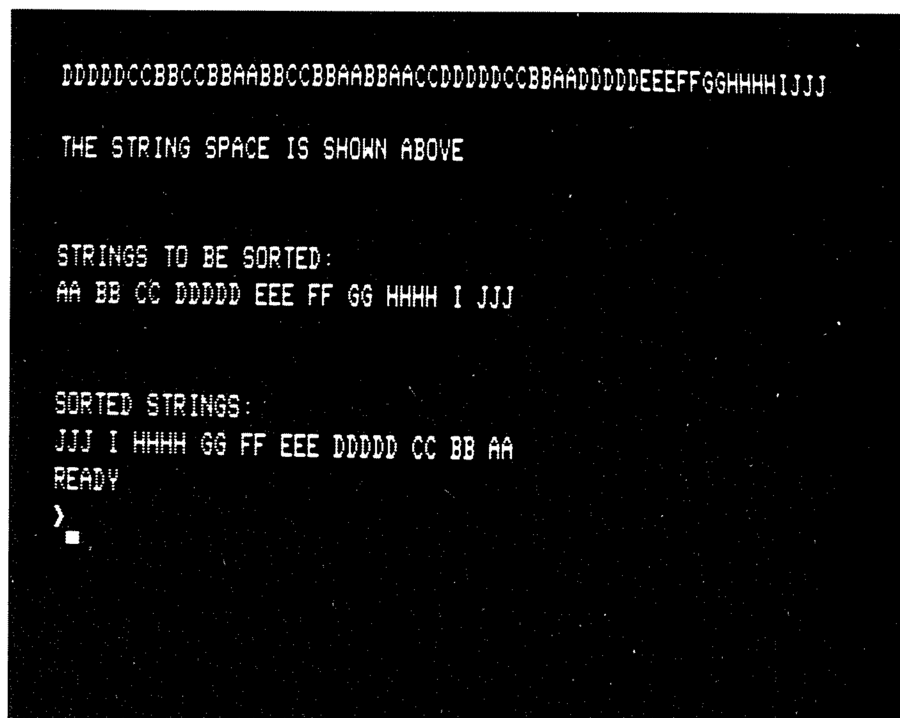


Photo 8-K

Chapter Summary

This chapter begins the study of **machine language**, the "native language" of the Z-80 processor, the heart of the TRS-80 Models I, III, and 4.

Programmers use an Editor to write programs in assembly language. Then they use an Assembler to convert these programs to machine language.

The following assembly language instructions were covered in this chapter:

ORG tells the assembler where the program is to start in memory.

LD loads a number from one memory location or register to another.

RET returns from the current routine to whatever called the routine (whether it was called from TRSDOS, BASIC, or another assembly language routine.

END ends the program and tells the assembler what point in the

program will be the first to execute.

INC increments (adds one) to whatever register or register pair follows the instruction.

CP compares the value in the A register to the value that follows the instruction.

JR "jumps" to the label or location that follows the instruction.

LDIR does the following repeatedly until BC=0:
-loads the value from the location pointed to by HL to the location pointed to by DE;
-increments (adds one to) DE;
-increments HL;
-decrements (subtracts one from) BC.

A machine language routine can be executed from within a BASIC program by protecting a certain area of memory, POKEing the object code into the protected memory, defining the USR call at the correct location, and calling the machine language routine with a USR call.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Define assembly language and its relationship to machine language.
B. Convert a decimal number to hexadecimal form.
C. Convert a hexadecimal number to decimal form.
D. Describe the uses in a Z-80 assembly language program of the following instructions: ORG, LD, INC, CP, JR, RET, END, and LDIR.
E. Given a listing of a assembled assembly language program, write a BASIC program that POKEs the assembly language program into memory and calls from BASIC.

More Programming Practice

Activities 1 to 4 require the use of an Editor Assembler.

1. Write an assembly language program that loads the reverse question mark (character 252) into video memory location 15488.

2. Write a BASIC program that calls the machine language routine produced in exercise 1 above.

3. Write an assembly language program that moves the first 1024 bytes to an area in memory starting with 55000D. Then move it from the area starting with 55000D to the video memory.

4. Write a program that examines each area of the video memory. If the program finds a character 32D (a space), it should load a character 191 into that location.

5. Using the BASIC program with the LDIR machine language subroutine presented in this chapter, write a program that asks the user for the move from, move to, and how many values. The program should then move the memory as specified.

Chapter Checkup

1. What is the relationship between assembly language and machine language programs?

2. Why are comments more important in assembly language programs than they are in BASIC programs?

3. What is source code in an assembly language program? What is its relationship to object code?

4. What is the hexadecimal value of 150D? What is the decimal value of FFA0H?

5. Which register is called the accumulator? Which register contains the flags?

6. What is the difference between the result of **LD HL,191** and **LD (HL),191**?

7. The partial program listing in Listing 8-J has an INKEY$ routine that checks for CHR$(10) and CHR$(91). Which keys return those ASCII values?

8. In Listing 8-J, which variable is incremented to change the part of memory viewed?

9. The description of line 370 in Listing 8-K refers to location 48000 as "the top of memory available to BASIC." Why is the memory above 48000 unavailable to BASIC in this program?

10. To you as a programmer, what is the most important advantage of assembly language programming compared to BASIC programming? What is the most important disadvantage of assembly language programming?

Chapter 9.
More Graphics in Assembly Langauge

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write an explanation of how an assembly language program that moves the screen sideways works.
B. Write an explanation of how an assembly language program that creates an alternate-character design on the screen works.
C. Write an explanation of how an assembly language program that splits the screen works.
D. Write an explanation of how an assembly language program that reverses the screen works.
E. Write an comparison between the BASIC SET instruction and the assembly language SET instruction.
F. Write an explanation of how an assembly language program that draws horizontal lines works.

INTRODUCTION

Chapter 8 provided an introduction to assembly language and machine language, as well as assembly language programs that fill the screen with characters and move values from one memory location to another very quickly. This chapter goes into more depth with assembly language, showing how to scroll part or all of the characters on the screen sideways, how to create a screen design, how to reverse all the graphic characters on the screen, and how to draw horizontal lines in assembly language.

SCROLLING THE SCREEN SIDEWAYS

We all have seen the characters on a computer screen scroll. Scrolling is simply moving all or part of the characters on the screen in one direction, and then off that side of the screen. When a BASIC program is too big to be displayed on the screen all at once, LISTing the program will cause it to scroll off the top of the screen. This kind of scrolling can be done with graphics characters as well as text. Listing 9-A shows an example of vertical scrolling of graphics.

```
10 CLS
20 FOR J=1 TO 200
30  PRINT @ RND(1023), CHR$(128+RND(63));
40 NEXT J
50 FOR J=1 TO 16
60  PRINT @ 1023, " "
70 NEXTJ
```

Listing 9-A

This program clears the screen, puts random graphics characters on the screen, and scrolls them off the top. The technique used in this vertical scroll is simply PRINTing a series of spaces on the last line. Each space PRINTed causes the screen to scroll one time.

Scrolling sideways is trickier. PRINTing on the screen in BASIC never causes this to happen. Therefore, to make the screen scroll sideways, we must write a new routine. To move all 1023 characters on the screen even one space over in a very short time would be impossible in BASIC, because BASIC is too slow. So we must call machine language to the rescue!

Sideways scrolling seems like a simple task in concept: moving data from one memory location to another is exactly what the LDIR instruction does. Let's consider how this might work.

You may recall the flexible LDIR program written in BASIC (Listing 8-I). By plugging in appropriate numbers we can do a sideways scroll of the screen with this program. Use these as the last four lines of the program:

```
200 MF = 15361: MT = 15360: HM = 1023
210 FOR I=1 TO 64
220   GOSUB 110
230 NEXT I
```

Listing 9-B

This program is easy to understand if you think of it as follows. The program moves a block of 1023 characters from a block starting with 15361 to a block starting with 15360. In other words, the whole block is moved one character to the left. Lines 210 to 230 repeat this block move 64 times, so the whole screen scrolls left.

This program has two problems. First, it is still slower than desired. Second, and perhaps more noticeably, this program wraps around from the left side of the screen to the right side. The character that was the first character of the second row becomes the last character of the first row. The effect is not one of scrolling off the left side of the screen, but rather of wrapping around.

The first problem can be partially solved quite easily. The subroutine beginning at line 110 (see Listing 8-I) calculates the values to POKE for the machine language routine each time it is called. Since these values are always the same (MF=15361, MT=15360, HM=1023), we can call the machine language routine directly the last 63 times without recalculating the POKE numbers.

```
205 GOSUB 110
210 FOR I=1 TO 63
220   X=USR1(0)
230 NEXT I
```

Listing 9-C

This improves the speed. But it does not eliminate the wrap-around problem. The desired effect is for the right side of the screen to remain blank as the characters scroll off the left side. This could be accomplished by POKEing a blank space to the first character of each row before each move. The block move would then move the blank character to the right side of the screen. But 15 or 16 POKEs before each move would slow down the routine too much. Let's consider an assembly language alternative.

```
                10  ;Left Scroll Program
                20          ORG   50000D      ;START POINT IN MEM
C350 00         30  BEGIN:  NOP               ;NO OPERATION: LABEL
C351 2188C3     40          LD    HL,COUNT    ;COUNT TO HL
C354 3600       50          LD    (HL),00     ;MAKE COUNT 00
C356 2188C3     60  LOOP0:  LD    HL,COUNT    ;COUNT TO HL
C359 7E         70          LD    A,(HL)      ;COUNT VALUE TO A
C35A FE40       80          CP    64D         ;COUNT = 64?
C35C 2829       90          JR    Z,DONE      ;IF YES, TO "DONE"
C35E 3C        100          INC   A           ;NEXT COUNT
C35F 77        110          LD    (HL),A      ;STORE IN "COUNT"
C360 CD65C3    120          CALL  START       ;BLANK & MOVE LOOP
C363 18F1      130          JR    LOOP0       ;BACK TO LOOP0 TIL 64
C365 00        140  START:  NOP               ;NO OPERATION: LABEL
C366 014000    150          LD    BC,64D      ;INC FOR VIDRAM ROWS
C369 21003C    160          LD    HL,15360D   ;BEG VIDRAM TO HL
C36C 7C        170  LOOP1:  LD    A,H         ;MSB OF VIDRAM TO A
C36D FE40      180          CP    40H         ;PAST VIDRAM?
C36F 2805      190          JR    Z,MOVE      ;IF YES, EXIT
C371 3620      200          LD    (HL),20H    ;SPACE TO VIDRAM
C373 09        210          ADD   HL,BC       ;NEXT ROW OF VIDRAM
C374 18F6      220          JR    LOOP1       ;AGAIN
C376 21FF3F    230  MOVE:   LD    HL,16383D   ;LAST CHAR OF VIDRAM
C379 3620      240          LD    (HL),20H    ;SPACE TO LAST CHAR
C37B 11003C    250          LD    DE,15360D   ;BEG. OF VIDRAM
C37E 21013C    260          LD    HL,15361D   ;DE+1
C381 01FF03    270          LD    BC,1023D    ;CHARS IN VIDRAM
C384 EDB0      280          LDIR              ;MOVE THEM!
C386 C9        290          RET               ;RETURN TO LINE 130
C387 C9        300  DONE:   RET               ;RETURN FROM ROUTINE
C388 0000      310  COUNT   DEFW  0000D       ;STORAGE: "COUNT"
C350           320          END   BEGIN
```

Listing 9-D

The program begins at 50000D, as specified by the ORG instruction in line 20. The rest of this assembly language program can be broken into parts with three functions: the counter, the blank line, and the move.

Lines 30 to 130 make up a counter which counts to 64, the number of characters in the horizontal line. This routine calls **START** (see line 120) 64 times. Here is a close look at how the counter works.

Line	Description
30	No operation. This line is used as a placeholder for the label. Other code can be inserted between the label (line 30) and the first instruction (line 40) if needed, without altering the label. For the most efficiency, the NOP would be removed in a final program.
40	Sets HL to point at the section of memory labelled **COUNT**. This memory is set aside in line 310.
50	Loads the memory pointed to by HL (which is **COUNT**) with 00. This initializes the counter at zero.
60	The first line of the loop labelled **LOOP0**. Each time through, the HL register pair is set to **COUNT**.
70	Loads the value pointed to by HL (which is **COUNT**) into register A.
80	Compares the value in A (which is still **COUNT**) with 64D.
90	If the value in A is 64 (and thus the "Z" flag is set), the program jumps to **DONE**, which returns to whatever called this program.
100	Increments A, which contains the counter.
110	Loads the incremented value back to **COUNT**.
120	Calls the **START** routine. The progam makes this call 64 times.
130	Jumps back to **LOOP0** for another time through the loop.

The **START** routine which line 120 calls consists of lines 140 to 290. This routine can be broken into two smaller routines.

Lines 140 to 220 place a vertical line of spaces on the leftmost column on the display. This makes possible the "scrolling off" effect, since the characters in the column of spaces are moved to the rightmost column of the screen, making it blank.

Line	Description
140	A no operation label like line 30.
150	The value 64D is loaded into register pair BC. This will later be added to the current screen value to move down 64 positions, to form a vertical column.
160	HL is loaded with 15360D, the first character of the video memory.
170	This instruction loads the most significant byte of the HL pair to A.
180	This value is compared to 40H, the most significant byte of the first memory location past the video memory.
190	If the value is 40 (which would occur when the HL value exceeded the values in the video memory), the program jumps to **MOVE**.
200	Otherwise, a character 20H (or 32D, a space) is loaded to that position of the video memory. This forms the line.
210	HL = HL + 64. This moves to the next line of the display.

220 The program loops back to LOOP1 for another pass.

The final section of the program does the actual move.

Line	Description
230	Loads the position of the last byte of the screen to HL.
240	Loads a space to the last position of the screen. This is so the last character of the screen will not leave a trailing character when the block move is made. If this position contained a visible character, that character would be repeated across the bottom row of the display.
250	DE, the "move to" register pair for LDIR, is set at the beginning of the video memory.
260	HL, the "move from" register pair for LDIR, is set at the next byte so the block move will move everything over one position.
270	BC, the "how many" register pair for LDIR, is set at 1023, the number of bytes on the screen minus one.
280	The block is moved.
290	The **START** routine RETurns to line 130.
300	The final RETurn. This is jumped to from line 90.
310	The label **COUNT** is defined as a "word," or a four-byte number. The value assigned is 00.
320	Here the program ends. The assembler is told to assemble the program to execute starting at the **BEGIN** label.

If you have an Editor - Assembler, you may key in this assembly language program and execute it. If you prefer to run this program from BASIC, the same loading and running routine used before will do fine. (Remember to protect memory at 48000.)

```
10 'MOVESCRN/BAS (Protect memory at 48000)
20 CLEAR500: GOTO110
30 FOR J=0 TO LEN(A$)/2-1
40  TE$ = MID$(A$,J*2+1,2)
50  MSB = ASC(LEFT$(TE$,1)): MSB = MSB-48:
          IF MSB>9 THEN MSB = MSB-7
60  LSB = ASC(RIGHT$(TE$,1)): LSB = LSB-48:
          IF LSB>9 THEN LSB = LSB-7
70  NU = MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 A$="002188C336002188C37EFE4028293C77CD65C318F100
01400021003C7CFE4028053620091 8F621FF3F362011003C
21013C01FF03EDB0C9C90000" 'SCRNMOVE/CMD
120 BG = 50000-2 16
130 DEFUSR1 = 50000-2 16
140 GOSUB30
150 FOR J=1 TO 100
```

```
160  POKE 15360+RND(1023), 127+RND(64)
170 NEXT J
180 X = USR1(0)
```

Listing 9-E

The hex form of the object code appears on line 110. This program loads the machine language program into memory at 50000, puts some graphics on the screen, and executes the machine language move program.

Photo 9-E-1

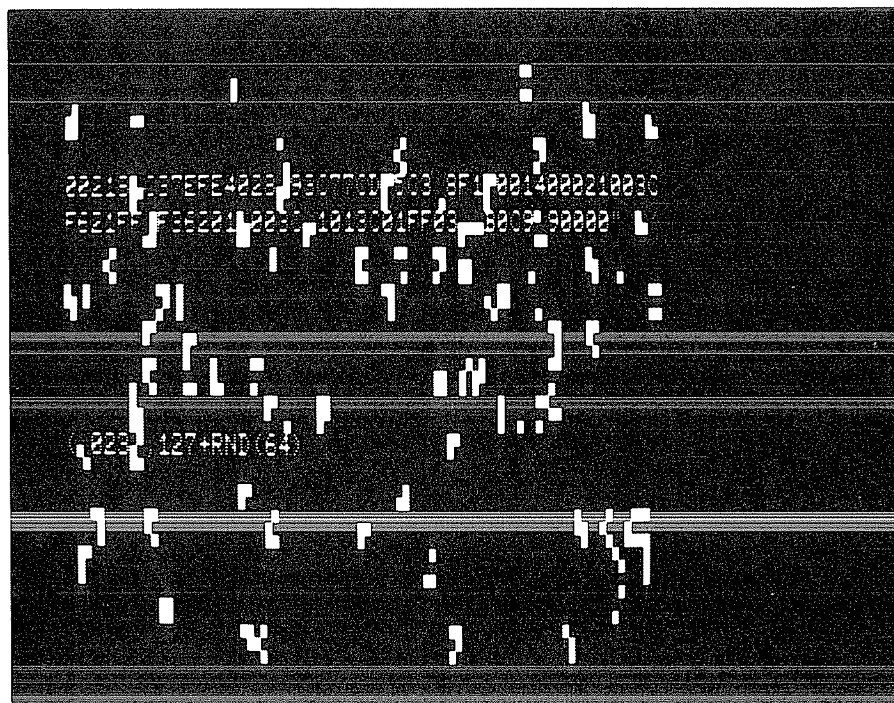


Photo 9-E-2

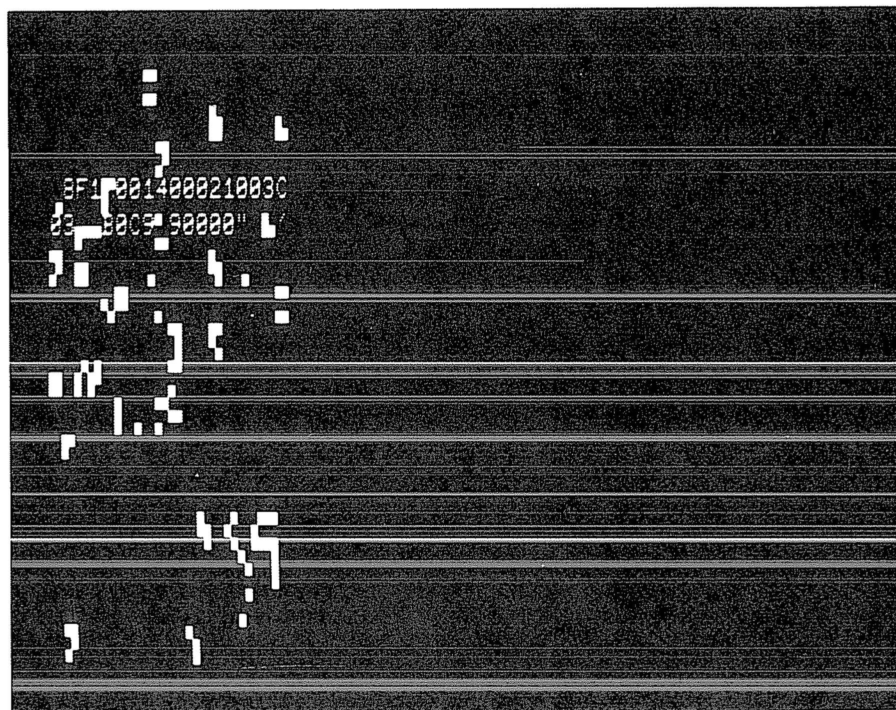


Photo 9-E-3

A MOVING DESIGN

The next program demonstrates the use if the IX register to create a moving design on the screen. The IX and IY registers are the only registers that can load with an offset, as seen in line 410 of the program. This program shows how to call ROM routines as a shortcut to programming. It also demonstrates the use of the PUSH and POP instructions.

```
                100  ;Design Maker by Dennis Tanner
                110  ;
0060            120  DELAY  EQU   0060H       ;ROM ROUTINE AT 0060H
028D            130  KBBRK  EQU   028DH       ;ROM ROUTINE AT 028DH
C738            140         ORG   51000D      ;START POINT IN MEM
C738  00        150  START: NOP               ;NO OPERATION: LABEL
C739  21FE3B    160         LD    HL,15358D   ;HL=VIDRAM-2
C73C  16BF      170         LD    D,191D      ;FIRST GRAPHIC CHAR
C73E  1E8C      180         LD    E,140D      ;SECOND GRAPHIC CHAR
C740  CD67C7    190         CALL  LOOP1       ;CALL GRAPHIC ROUTINE
C743  010080    200         LD    BC,8000H    ;VALUE FOR DELAY
C746  CD6000    210         CALL  DELAY       ;WAIT A WHILE
C749  CD8D02    220         CALL  KBBRK       ;CHECK BREAK KEY
C74C  C279C7    230         JP    NZ,DONE     ;IF PRESSED, TO DONE
C74F  21FE3B    240         LD    HL,15358D   ;HL=VIDRAM-2
C752  168C      250         LD    D,140D      ;FIRST GRAPHIC CHAR
C754  1EBF      260         LD    E,191D      ;SECOND GRAPHIC CHAR
C756  CD67C7    270         CALL  LOOP1       ;CALL GRAPHIC ROUTINE
C759  010080    280         LD    BC,8000H    ;VALUE FOR DELAY
C75C  CD6000    290         CALL  DELAY       ;WAIT A WHILE
C75F  CD8D02    300         CALL  KBBRK       ;CHECK BREAK KEY
C762  C279C7    310         JP    NZ,DONE     ;IF PRESSED, TO DONE
C765  18D1      320         JR    START       ;ONE MORE TIME!
C767  23        330  LOOP1: INC   HL          ;NEXT SCREEN POSITION
C768  23        340         INC   HL          ;NEXT SCREEN POSITION
C769  7C        350         LD    A,H         ;TO COMPARE MSB
C76A  FE40      360         CP    40H         ;BEYOND VIDRAM?
C76C  280B      370         JR    Z,DONE      ;IF YES, TO DONE
C76E  E5        380         PUSH  HL          ;TO TRANSFER FROM HL
C76F  DDE1      390         POP   IX          ;TO IX
C771  DD7200    400         LD    (IX),D      ;D VALUE TO VIDRAM
C774  DD7301    410         LD    (IX+1),E    ;E VALUE TO NEXT POS
C777  18EE      420         JR    LOOP1       ;ONE MORE TIME
C779  C9        430  DONE:  RET               ;RETURN
C738            440         END   START       ;END PROGRAM
```

Listing 9-F

Lines 120 and 130 define the labels "DELAY" AND "KBBRK" as locations 0060H and 028DH respectively. These locations are in the ROM, the part of memory that is always there. The TRS-80 Model I, III, and 4 ROM has certain locations that can be called to perform certain functions. (See the TRS-80 Operation and BASIC Langauge Reference Manual for a complete list.)

The DELAY routine simply delays for a specified interval. The programmer specifies the interval by loading the BC register pair. In this case, BC is loaded with 8000H before the CALL to DELAY is made. 8000H tells the routine to delay for about one-half second.

The KBBRK routine scans the keyboard one time and looks for the BREAK key. If the BREAK key is held down as the scan is made, the routine returns a NZ (nonzero) flag. Thus lines 230 and 310 jump to the "DONE" address if the BREAK key is pressed **(JP NZ,DONE)**.

This program might be broken into two functional parts, a **calling, delaying, and looping** section (lines 150 to 320) and a **graphics loading** section (lines 330 to 420). Let's take a close look at these sections. First, the looping section.

Line	Description
150	No operation. This location is used only as a label.
160	The beginning location of the video memory ("VIDRAM") minus two is loaded into HL. This initializes the HL pair which will be incremented two at a time in lines 330 and 340.
170	Loads one graphics character value into register D.
180	Loads a second graphics character value into register E. These values in D and E will be loaded into the video memory in lines 400 and 410.
190	Calls the routine that puts the graphics on the screen, using the LOOP1 label. This routine starts on line 330.
200	Loads 8000H into BC. This sets the length of the delay at about one-half second when the DELAY routine is called in the next line.
210	The DELAY routine is called.
220	The KBBRK routine checks whether the user is holding down the BREAK key.
230	If the BREAK key is being held down, and the Z flag is in the "NZ" state, the routine jumps to the DONE label (line 430).

Lines 240 to 310 repeat the steps that appear in lines 160 to 230, except the D and E values are switched. The result is that the pattern appears to reverse.

Line 320 jumps back to the START label (line 150) to repeat both kinds of patterns.

Thus, the first part of the program repeatedly loads registers with certain values, calls the graphics routine, delays, checks the BREAK key, and repeats the process with the graphics characters exchanged.

The second part of the program (lines 330 to 420) puts the graphics design on the screen.

Line	Description
330	Increments HL. Since HL points to the current video memory location, this line points HL to the next video memory location.
340	Same as 330. The video memory pointer is thus incremented by 2.
350	To determine whether the video memory pointer has reached the end of the video memory, the most significant byte (MSB) of the pointer is loaded into A.
360	This byte is compared with 40H, the MSB of the area just beyond the video memory.
370	If the value is beyond the video memory, the program jumps to the label "DONE" which RETurn to the routine that called it (see lines 190 and 270 above).
380	In this program, the characters are loaded into the video memory locations pointed to by register IX and IX+1. The program therefore must transfer the value in HL (the video memory pointer) to IX. Since no **LD IX,HL** instruction exists in the Z-80 instruction set, the HL value is PUSHed onto the stack. (See next line.)
390	The video memory value just PUSHed onto the stack, is POPped off the stack and loaded into the IX register, completing the transfer of the value from HL to IX.
400	Now the memory location pointed to by IX is loaded with the value in the D register, making one graphics character appear on the display.
410	The next location (IX+1) is loaded with the other graphics character value, the one in the E register. This makes the other character appear on the display.
420	The program jumps back to the "LOOP1" label for the next memory location.

In this program, the IX register pair was chosen to demonstrate a special capability of this register pair: indexed addressing. The IX and IY register pairs are the only ones which permit addressing with an offset, like **LD (IX+1),E** as is used in this program.

The total effect of this program is to display a screen which shifts back and forth between two character patterns, pausing and checking the BREAK key after each display.

SPLITTING THE SCREEN

We have seen a left-scrolling program and a design-making program so far in this chapter. The next program shows how to split the screen, moving the left side of the screen to the left and the right side to the right.

```
                100 ;Screen splitter SPLIT/CMD
                110 ;
CB20            120         ORG   52000D         ;START POINT IN MEM
CB20 CD27CB     130 START:  CALL  MIDLN          ;MAKE LINE IN MIDDLE
CB23 CD4BCB     140         CALL  SPLIT          ;SPLIT SCREEN
CB26 C9         150         RET                  ;RETURN TO CALLER
CB27 2185CB     160 MIDLN:  LD    HL,LINECT      ;LOAD LINE COUNTER
CB2A 36E0       170         LD    (HL),0E0H      ;LOAD W/LSB VR-32
CB2C 23         180         INC   HL             ;NEXT POS (LINECT)
CB2D 363B       190         LD    (HL),3BH       ;LOAD W/MSB VR-32
CB2F 3A86CB     200 LNLP1:  LD    A,(LINECT+1)   ;LOAD LINE COUNTER
CB32 67         210         LD    H,A            ;MSB OF LINECT TO H
CB33 3A85CB     220         LD    A,(LINECT)     ;LOAD LINE COUNTER
CB36 6F         230         LD    L,A            ;LSB OF LINECT TO L
CB37 014000     240         LD    BC,64D         ;TO INC HL
CB3A 09         250         ADD   HL,BC          ;NEXT VIDRAM LINE
CB3B 7C         260         LD    A,H            ;LD A W/MSB OF VR
CB3C FE40       270         CP    40H            ;PAST VIDRAM?
CB3E C8         280         RET   Z              ;IF YES, RETURN
CB3F 3620       290         LD    (HL),20H       ;SPACE INTO MIDDLE
CB41 7D         300         LD    A,L            ;LSB OF LINECT
CB42 3285CB     310         LD    (LINECT),A     ;PUT BACK TO LINECT
CB45 7C         320         LD    A,H            ;MSB OF LINECT
CB46 3286CB     330         LD    (LINECT+1),A   ;PUT BACK TO LINECT
CB49 18E4       340         JR    LNLP1          ;FOR NEXT LINE
CB4B 2183CB     350 SPLIT:  LD    HL,SPCT        ;MOVE CTR TO HL
CB4E 3621       360         LD    (HL),33D       ;SET CTR AT 33
CB50 2183CB     370 SPLOP:  LD    HL,SPCT        ;LOAD CTR INTO HL
CB53 7E         380         LD    A,(HL)         ;COUNTER TO A
CB54 3D         390         DEC   A              ;CTR=CTR-1
CB55 77         400         LD    (HL),A         ;PUT CTR BACK
CB56 FE00       410         CP    0              ;DONE W/COUNTER?
CB58 C8         420         RET   Z              ;IF YES, RETURN
CB59 CD5ECB     430         CALL  SPLP0          ;SPLIT ONE SPACE
CB5C 18F2       440         JR    SPLOP          ;GO ONCE MORE
CB5E 21FF3B     450 SPLP0:  LD    HL,15359D      ;VIDRAM-1
CB61 010100     460 SPLP1:  LD    BC,1D          ;TO ADD 1 TO HL
CB64 09         470         ADD   HL,BC          ;HL=HL+1
CB65 7C         480         LD    A,H            ;MSB OF HL VIDRAM
CB66 FE40       490         CP    40H            ;BEYOND VIDRAM?
CB68 C8         500         RET   Z              ;IF YES, RETURN
CB69 E5         510         PUSH  HL             ;TO LOAD TO DE
CB6A D1         520         POP   DE             ;HL TO DE
CB6B 09         530         ADD   HL,BC          ;HL=HL+1 FOR LDIR
CB6C 012000     540         LD    BC,32D         ;COUNTER FOR HL
CB6F EDB0       550         LDIR                 ;MOVE 'EM LEFT!
CB71 011E00     560         LD    BC,30D         ;TO ADD TO HL
CB74 09         570         ADD   HL,BC          ;HL=HL+30 (EOL)
CB75 E5         580         PUSH  HL             ;TO LOAD INTO DE
CB76 D1         590         POP   DE             ;HL TO DE
CB77 28         600         DEC   HL             ;HL=DE-1 AT EOL
```

```
CB78 012000   610          LD    BC,32D       ;COUNTER FOR LDDR
CB7B EDB8     620          LDDR               ;MOVE 'EM RIGHT!
CB7D 012100   630          LD    BC,33D       ;TO ADD TO HL
CB80 09       640          ADD   HL,BC        ;ADD 33: TO NEXT LINE
CB81 18DE     650          JR    SPLP1        ;AGAIN
CB83 0000     660 SPCT     DEFW  0000H        ;CTR FOR MOVE
CB85 0000     670 LINECT   DEFW  0000H        ;CTR FOR LINES
CB20          680          END   START        ;END PROGRAM
```

Listing 9-G

This program has three main parts:

Lines 130 to 150 call the other two main routines and RETurn.

Lines 160 to 340 draw a line down the center of the screen.

Lines 350 to 650 repeat 33 times a routine which moves the left half of the screen left one character, and the right half of the screen right one character.

Rather than look at a line-by-line description, let's look at some of the new techniques used in this program.

o A four-byte "word" is defined as the line counter, and another is defined as the split counter (lines 660 and 670). All four bytes of the LINECT "word" are used, so both LINECT and LINECT+1 are used when storing and retrieving its value. Only the first two bytes of the SPCT "word" are used, and they are addressed directly as SPCT.

o Just as a line was drawn down the left side of the screen before the LDIR in the program in Listing 9-D, this program draws a line down the middle of the screen before doing its block moves. If this were not done, both the left and the right block moves would leave trailing characters.

o The left half of the screen is moved with an LDIR instruction (line 550). The right half of the screen is moved with a LDDR instruction. This LDDR instruction works the same as the LDIR except the DE and HL register pairs are decremented instead of being incremented. Just as the left side of the screen is moved with the left-to-right (incrementing) action of the LDIR instruction, the right side of the screen is moved with the right-to-left (decrementing) action of the LDDR instruction.

The last two programs (the design-maker and the screen splitter) can be loaded into memory via the following BASIC program. Notice that the design program loads into memory beginning at 51000, and the screen splitter loads in beginning at 52000. (Remember to protect memory at 48000 when running this program.)

```
10 'DESIGN/BAS Combining the design program and the screen splitter
20 CLEAR 500: GOTO 110
30 FOR J=0 TO LEN(A$)/2-1
40  TE$ = MID$(A$,J*2+1,2)
50  MSB = ASC(LEFT$(TE$,1)): MSB = MSB - 48:
      IF MSB>9 THEN MSB = MSB - 7
60  LSB = ASC(RIGHT$(TE$,1)): LSB = LSB - 48:
      IF LSB>9 THEN LSB = LSB - 7
70  NU = MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 A$="0021FE3B16BF1E8CCD67C7010080CD6000CD8D02C279C
      721FE3B168C1EBFCD67C7010080CD6000CD8D02C279C718D1
      23237CFE40280BE5DDE1DD7200DD730118EEC9" ' DESIGN/CMD
120 BG = 51000 - 2 16
130 DEFUSR1 = BG
140 GOSUB 30
150 A$="CD27CBCD4BCBC92185CB36E023363B3A86CB673A85CB
      6F014000097CFE40C836207D3285CB7C3286CB18E42183CB36
      212183CB7E3D77FE00C8CD5ECB18F221FF3B010100097CFE4
      0C8E5D109012000EDB0011E0009E5D12B012000EDB801210009
      18DE00000000" 'SPLIT/CMD
160 BG = 52000 - 2 16
170 DEFUSR2 = BG
180 GOSUB 30
190 CMD"B","OFF"
200 X = USR1 (0)
210 CMD"B","ON"
220 X = USR2 (0)
230 FOR J=1 TO 50
240  I$ = INKEY$
250  IF I$<>"" THEN END
260 NEXTJ
270 GOTO190
```

Listing 9-H

This program uses the same subroutine to POKE the machine language programs into memory. Line 110 contains the hex version of the object code for the design program, and line 150 contains the code for the screen splitter. USR call 1 points to the design program, and USR call 2 points to the splitter.

After the call to the BASIC subroutine that POKEs the values into memory (line 180), the **BREAK** key is locked out. Since the machine language design program loops continually until the **BREAK** key is pressed, the user must press that key to return from the design program. Of course, pressing **BREAK** exits from a BASIC program, so the key was locked out to permit returning from the machine language program without stopping the BASIC program. The **BREAK** key is enabled after

the design program executes, in line 210.

USR call 2 (line 220) causes the characters on the display to scroll to the sides, giving the appearance of splitting the screen. Lines 230 to 260 add a way of exiting the program since the BREAK key is disabled during the execution of the machine language programs, as well as between lines 190 and 210.

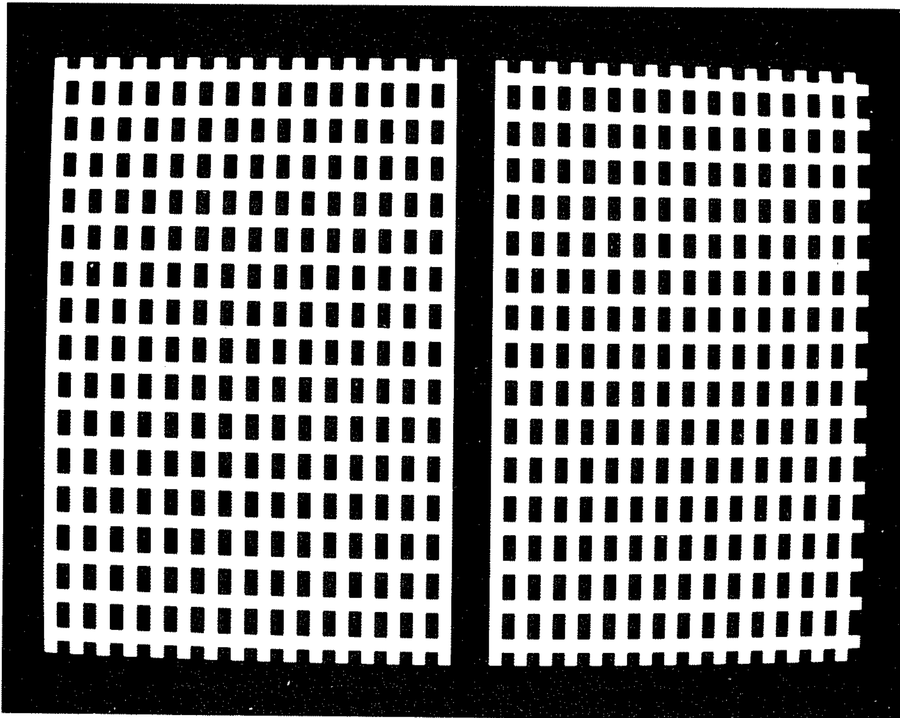


Photo 9-G-1

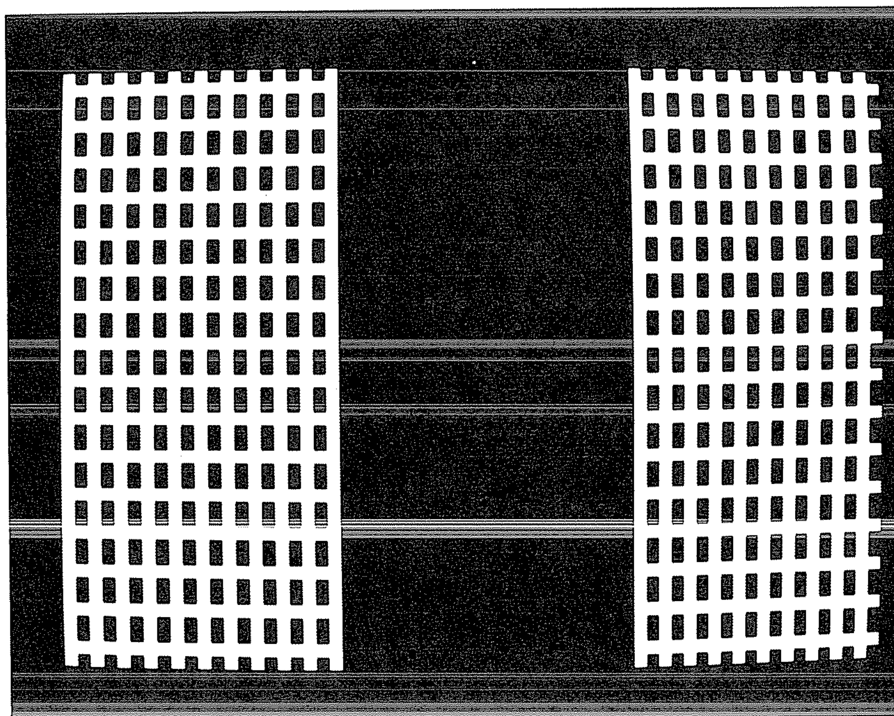


Photo 9-G-2

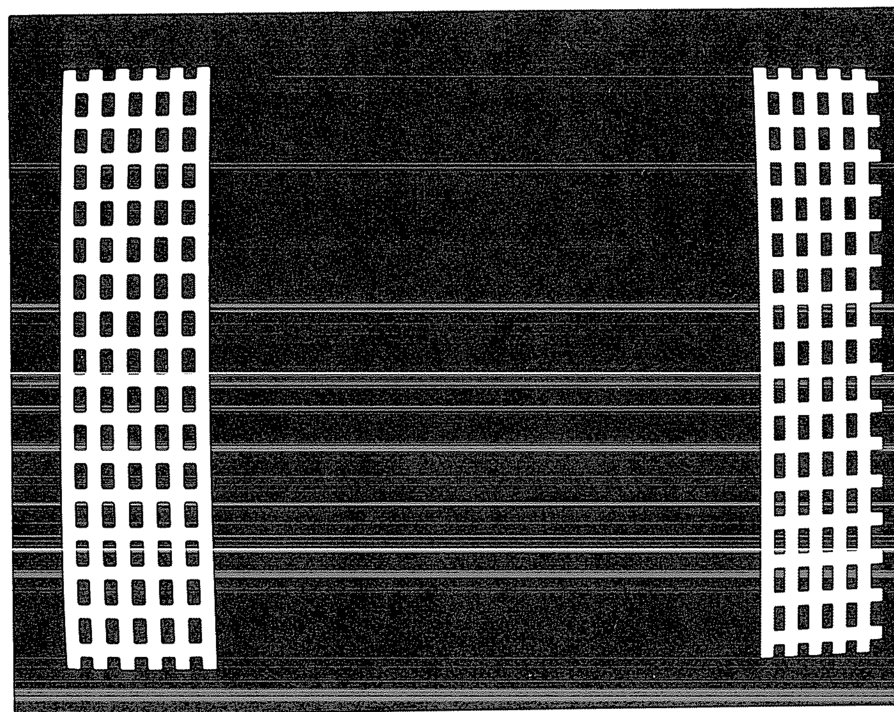


Photo 9-G-3

REVERSING THE GRAPHICS CHARACTERS ON THE SCREEN

The next sample program alters rather than moves the characters on the screen. It checks each character on the screen to be sure it is a graphics character. If it is, the program subtracts the value of the current character from a certain number and loads that back into the video memory. Each graphics character is reversed. If a screen position starts out with only the upper left corner "lit", it will end up with every position except the upper left corner lit.

```
                100 ; Reverse Program
                110 ;
CF00            120          ORG   53000D      ;START POINT IN MEM
CF00 00         130 START:   NOP               ;NO OPERATION: LABEL
CF09 21FF3B     140          LD    HL,15359D   ;VIDRAM - 1
CF0C 23         150 LOOP1:   INC   HL          ;NEXT VIDRAM POS
CF0D 7C         160          LD    A,H         ;MSB OF LOC
CF0E FE40       170          CP    40H         ;BEYOND VIDRAM?
CF10 280D       180          JR    Z,DONE      ;IF YES, TO "DONE"
CF12 7E         190          LD    A,(HL)      ;SCRN CONTENTS TO A
CF13 FE7F       200          CP    127D        ;<127? NONGRAPHIC?
CF15 38F5       210          JR    C,LOOP1     ;IF YES, NEXT POS
CF17 7E         220 LOOP2:   LD    A,(HL)      ;SCRN CONTENTS TO A
CF18 47         230          LD    B,A         ;SCRN CONTENTS TO B
CF19 3E3F       240          LD    A,63D       ;SUBTRACTER TO A
CF1B 90         250          SUB   A           ;SUBTRACT SCRN VALUE
CF1C 77         260          LD    (HL),A      ;CHAR BACK TO SCREEN
CF1D 18ED       270          JR    LOOP1       ;ONE MORE TIME!
CF1F C9         280 DONE:    RET               ;RETURN TO CALLER
CF08            290          END   START       ;END PROGRAM
```

Listing 9-I

Lines 100 to 140 label the program with a comment, set the point in memory where the program will be assembled, and initialize HL at 15359D, the location immediately before the beginning of video memory. HL in incremented each time through the loop, so it must start out one less than the desired beginning location.

Lines 150 to 180 increment HL, which contains the current video memory location. They then check to see if HL is past the end of the video memory. If it is, the machine language program will RETurn to the calling program.

Lines 190 to 210 check to see if the character in the current video memory position is 127 or less, and is thus a text character. If it is, the program skips the reversing process and loops back for another character.

Lines 220 to 240 load the value of the current screen location to B, and load 63 to A. 63 is used because it is the two-byte equivalent to 319. (319 - 256 = 63.) 319 is the number from which any graphics

character can be subtracted to arrive at that character's reverse character. The clearest example of this is 319 - 191 = 128 (191 is the character with all bits "turned on," and 128 has all bits "turned off."

Lines 250 to 270 subtract B from A, load the resultant character back into the video memory, and loop back for the next screen position.

Lines 280 to 290 RETurn to the calling program, and end the program.

This program reverses any character with a value of 128 or greater. What if the character is a text space, character 32D? The program will not reverse the text space, so text on the screen looks normal (spaces included). But that means that if you want to clear the screen, do graphics, and then reverse the screen, you must clear the screen with a graphics space (character 128) for the program to reverse the spaces.

DRAWING LINES IN ASSEMBLY LANGUAGE

In Chapter 1 we saw how to draw lines in BASIC with the SET instruction. Assembly language uses the same word **SET** for its instruction that changes one bit in one memory location, but its format and usage is different.

Assembly language's SET instruction changes any bit in any available memory location, not just the video memory. This gives the programmer a lot of flexibility when manipulating numbers and other data. For this graphics program, we will restrict the use of the SET instruction to locations within the video memory.

To SET a point on the video display in assembly language, be sure BIT 7 of the location is SET and BIT 6 is not SET. (That assures it will be between 128 and 191.) Then SET the appropriate bits, using 0 for the top left, 1 for the top right, 2 for the middle left, 3 for the middle right, 4 for the bottom left, and 5 for the bottome right. Lines 430 to 560 execute the SET instructions in this routine.

```
                100 ;                                  ;
                110 ; Horizontal Line Drawer by Dennis Tanner
                120 ;                                  ;
D2F0            130          ORG    54000D          ;START POINT IN MEM
D2F0  00        140 START:   NOP                    ;NO OPERATION -- LABEL
D2F1  2157D3    150          LD     HL,54103D       ;LENGTH OF LINE
D2F4  7E        160          LD     A,(HL)          ;LENGTH INTO A
D2F5  57        170          LD     D,A             ;LENGTH INTO D
D2F6  2154D3    180          LD     HL,54100D       ;LSB OF LOC
D2F9  7E        190          LD     A,(HL)          ;INTO A
D2FA  4F        200          LD     C,A             ;INTO C
D2FB  2155D3    210          LD     HL,54101D       ;MSB OF LOC
D2FE  7E        220          LD     A,(HL)          ;INTO A
```

```
D2FF 47      230          LD    B,A        ;INTO B
D300 60      240          LD    H,B        ;MSB OF LOC TO H
D301 69      250          LD    L,C        ;LSB OF LOC TO L
D302 3E00    260          LD    A,0        ;TO USE AS LEN CTR
D304 5F      270          LD    E,A        ;STORE LEN
D305 60      280 LOOPA:   LD    H,B        ;MSB OF POS
D306 69      290          LD    L,C        ;LSB OF POS
D307 CB7E    300          BIT   7,(HL)     ;GRAPHICS CHAR?
D309 2003    310          JR    NZ,LOOP1   ;IF YES, TO LOOP1
D30B 3E80    320          LD    A,128D     ;IF TEXT, LD W/SPACE
D30D 77      330          LD    (HL),A     ;TO SCREEN
D30E 2156D3  340 LOOP1:   LD    HL,54102D  ;1, 2, OR 3
D311 7E      350          LD    A,(HL)     ;CHAR TYPE TO A
D312 F301    360          CP    1          ;TOP ROW?
D314 2809    370          JR    Z,TOP      ;IF YES, TO TOP
D316 FE02    380          CP    2          ;MIDDLE ROW?
D318 2810    390          JR    Z,MIDDLE   ;IF YES, TO MIDDLE
D31A FE03    400          CP    3          ;BOTTOM ROW?
D31C 2817    410          JR    Z,BOTTM    ;IF YES, TO BOTTOM
D31E C9      420          RET              ;OTHERWISE, RETURN
D31F CD4BD3  430 TOP:     CALL  GTPOS      ;PUT POS IN HL
D322 CBFE    440          SET   7,(HL)     ;TO MAKE GRAPHICS CHR
D324 CBC6    450          SET   0,(HL)     ;UPPER LEFT PIXEL
D326 CBCE    460          SET   1,(HL)     ;UPPER RT PIXEL
D328 1814    470          JR    LOOPB      ;PAST MID & BOTTOM
D32A CD4BD3  480 MIDDLE:  CALL  GTPOS      ;PUT POS IN HL
D32D CBFE    490          SET   7,(HL)     ;TO MAKE GRAPHICS CHR
D32F CBD6    500          SET   2,(HL)     ;SECOND ROW LT PIXEL
D331 CBDE    510          SET   3,(HL)     ;SECOND ROW RT PIXEL
D333 1809    520          JR    LOOPB      ;PAST MID & BOTTOM
D335 CD4BD3  530 BOTTM:   CALL  GTPOS      ;PUT POS IN HL
D338 CBFE    540          SET   7,(HL)     ;TO MAKE GRAPHICS CHR
D33A CBE6    550          SET   4,(HL)     ;BOTTOM LT PIXEL
D33C CBEE    560          SET   5,(HL)     ;BOTTOM RT PIXEL
D33E 7B      570 LOOPB:   LD    A,E        ;GET LEN BACK
D33F 3C      580          INC   A          ;INCRE LEN COUNTER
D340 5F      590          LD    E,A        ;PUT LEN BACK TO E
D341 23      600          INC   HL         ;INCRE SCREEN POS
D342 44      610          LD    B,H        ;STORE POSITION MSB
D343 4D      620          LD    C,L        ; & LSB
D344 F5      630          PUSH  AF         ;SAVE FLAGS
D345 BA      640          CP    D          ;A = DESIRED LEN?
D346 2806    650          JR    Z,DONE     ;IF YES, RETURN
D348 F1      660          POP   AF         ;RESTORE FLAGS
D349 18BA    670          JR    LOOPA:     ;OTHERWISE AGAIN
D34B 60      680 GTPOS:   LD    H,B        ;MSB OF POS
D34C 69      690          LD    L,C        ;LSB OF POS
D34D C9      700          RET              ;RET FROM GTPOS
```

```
D34E F1       710 DONE   POP   AF          ;RESTORE FLAGS
D34F C9       720        RET               ;RETURN FROM PROGRM
D2F0          730        END   START       ;END LISTING
```

Listing 9-J

This assembly language program is designed to be called by a BASIC program. Before calling the machine language program, the BASIC program POKEs values into certain memory locations. These values are read from the same memory locations by the machine language program. The memory locations included and the significance of the values in those locations are as follows:

Location	Function
54100	Least significant byte of line start location (video memory)
54101	Most significant byte of line start location
54102	Pixel flag (1 for top, 2 for middle, 3 for bottom)
54103	Length of desired line

The main body of this program can be broken down into six parts:

Lines 140 to 270 read the values from the various memory locations and store them in registers. Remember that these values were to have been POKEd into these locations by the BASIC program before the machine language program is called. Lines 260 and 270 set the "current length of line" counter (register E) at 0.

Lines 280 to 330 use the BIT instruction to check whether bit 7 of the current video memory character is set. If it is set, the character is 128 or above and the program proceeds. If it is not set, the character is a text character (or at least a character smaller than 128). In this case, lines 320 and 330 load a character 128 into the position. This assures that as the other bits are set, the resulting character will be a graphics character.

Lines 340 to 420 load the value that tells which pixel will be set (1 = top, 2 = middle, 3 = bottom) and branches to the routines below accordingly.

Lines 430 to 560 are the routines which actually set the pixels on the video screen. Each routine in this section (TOP, MIDDLE, BOTTM) calls the routine GETPOS which loads the current screen position into HL. Each routine then sets the appropriate bits for that section of the line, then jumps to LOOPB.

Lines 570 to 670 contains LOOPB. These lines compare the value stored in E (the current length of the line) to the value in D (the desired length of the line) to determine whether the line is long enough yet. If it is, the program returns. If not, it returns to line 280 (LOOPA). If the line is long enough, the program exits to line 710 (DONE).

Lines 680 to 700 load the current position into HL so the bits of it can be set (see lines 430 to 560).

The program thus executes START; LOOPA; LOOP1; one of TOP, MIDDLE, OR BOTTM; LOOPB; and back to LOOPA, until line is the desired length.

The BASIC program in Listing 9-K incorporates the machine language program above. (Remember to protect memory at 48000 when running this program.)

```
10 'LINE/BAS By Dennis Tanner
20 CLEAR 500: GOTO 110
30 FOR J=0 TO LEN(A$)/2-1
40  TE$ = MID$(A$,J*2+1,2)
50  MSB = ASC(LEFT$(TE$,1)): MSB = MSB-48:
          IF MSB>9 THEN MSB = MSB-7
60  LSB = ASC(RIGHT$(TE$,1)): LSB = LSB-48:
          IF LSB>9 THEN LSB = LSB-7
70  NU = MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 A$="002157D37E572154D37E4F2155D37E4760693E005F6069C
    B7E20033E80772156D37EFE012809FE022810FE032817C9CD4B
    D3CBFECBC6CBCE1814CD4BD3CBFECBD6CBDE1809CD4BD3
    CBFECBE6CBEE7B3C5F23444DF5BA2806F118BA6069C9F1C9"
120 'THAT WAS THE LINE DRAWING ROUTINE
130 BG = 54000 - 2 16: GOSUB30: DEFUSR1 = 54000 - 2 16
140 POKE 54100 - 2 16, &H00 'LSB OF VIDEO MEM POSITION
150 POKE 54101 - 2 16, &H3C 'MSB OF VIDEO MEM POSITION
160 POKE 54102 - 2 16, 2 'MIDDLE PIXELS IN CHARACTER
170 POKE 54103 - 2 16, 60 'LENGTH OF LINE
180 CLS: X = USR1(0): PRINT @ 128, ""
```

Listing 9-K

Once again, the hex version of the object code is assigned to A$ (line 110) and the subroutine beginning in line 30 POKEs it into memory. The USR address is assigned (line 130), and the beginning point, pixel specification, and length are POKEd into the proper positions (lines 140 to 170). In line 180 the screen is cleared, and the USR routine is called to draw the line.

The program in Listing 9-L includes the line drawing routine, the reverse routine, and a routine that clears the screen with a graphics space (character 128).

```
10 'LINE/BAS By Dennis Tanner (Protect memory at 48000)
20 CLEAR 500: GOTO 110
30 FOR J=0 TO LEN(A$)/2-1
40  TE$ = MID$(A$,J*2+1,2)
50  MSB = ASC(LEFT$(TE$,1)): MSB = MSB-48:
          IF MSB>9 THEN MSB = MSB-7
60  LSB = ASC(RIGHT$(TE$,1)): LSB = LSB-48:
          IF LSB>9 THEN LSB = LSB-7
70  NU = MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 A$="002157D37E572154D37E4F2155D37E4760693E005F6069C
   B7E20033E80772156D37EFE012809FE022810FE032817C9CD4B
   D3CBFECBC6CBCE1814CD4BD3CBFECBD6CBDE1809CD4BD3
   CBFECBE6CBEE7B3C5F23444DF5BA2806F118BA6069C9F1C9"
120 'THAT WAS THE LINE DRAWING ROUTINE
130 BG = 54000 - 2 16: GOSUB30: DEFUSR1 = 54000 - 2 16
140 A$ = "0021FF3B237CFE40280D7EFE7F38F57E473E3F907718EDC9"
          'THAT WAS THE REVERSE ROUTINE
150 BG = 53000-2 16: GOSUB30: DEFUSR2 = 53000-2[16
160 A$ = "21FF3B237CFE40C8368018F7"
          'THIS IS THE CLS ROUTINE WITH A GRAPHIC BLANK (80H)
170 BG = 49000 - 2 16: GOSUB30: DEFUSR3 = 49000-2 16
180 X = 5+RND(3): Y = 15+RND(8): Z = 5+RND(10)
190 X1 = USR3(0)
200 FOR J=0 TO 3
210  FOR K=0 TO 3
220   POKE 54101 - 2 16, &H3C+J
230   FOR L=1 TO 3
240    POKE 54100 - 2 16, K*64 + SIN((J*16+K*4+L)/X) * Y + 28
250    POKE 54103 - 2 16, Z
260    POKE 54102 - 2 16, L
270    X1 = USR1(0)
280   NEXT L
290  NEXT K
300 NEXT J
310 FOR ZX = 1 TO 11
320   X1 = USR2(0)
330   FOR ZZ=1 TO 50: NEXT ZZ
340 NEXT ZX
350 GOTO 180
```

Listing 9-L

This program loads the three programs into the high memory, then clears the screen with a graphics space (line 190). It then begins a loop which draws horizontal lines on the screen. In this loop (lines 200 to 300), J represents the most significant byte of the screen position, K

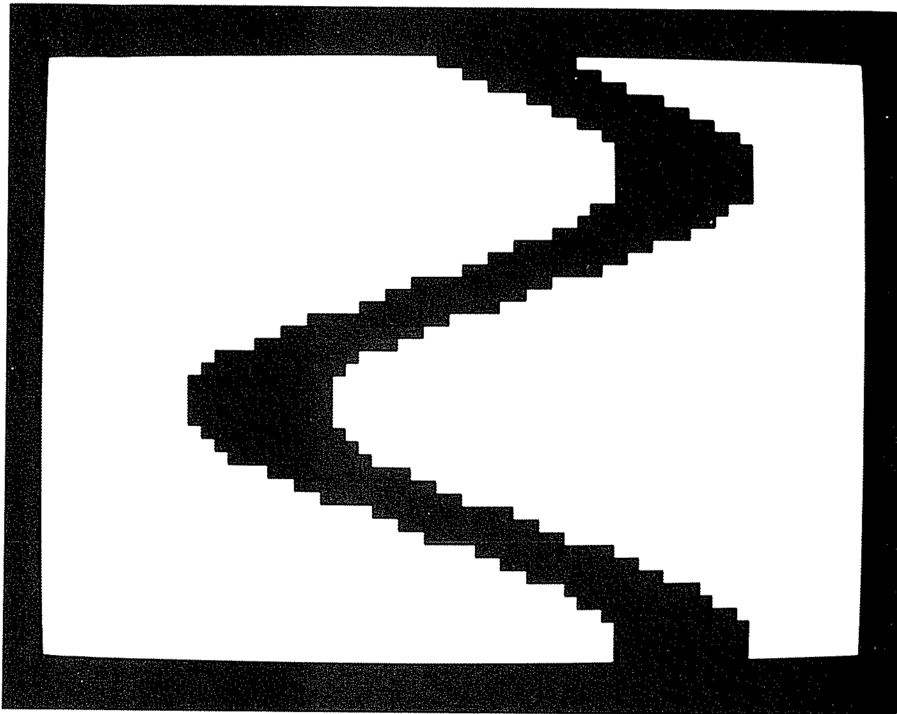


Photo 9-L-1

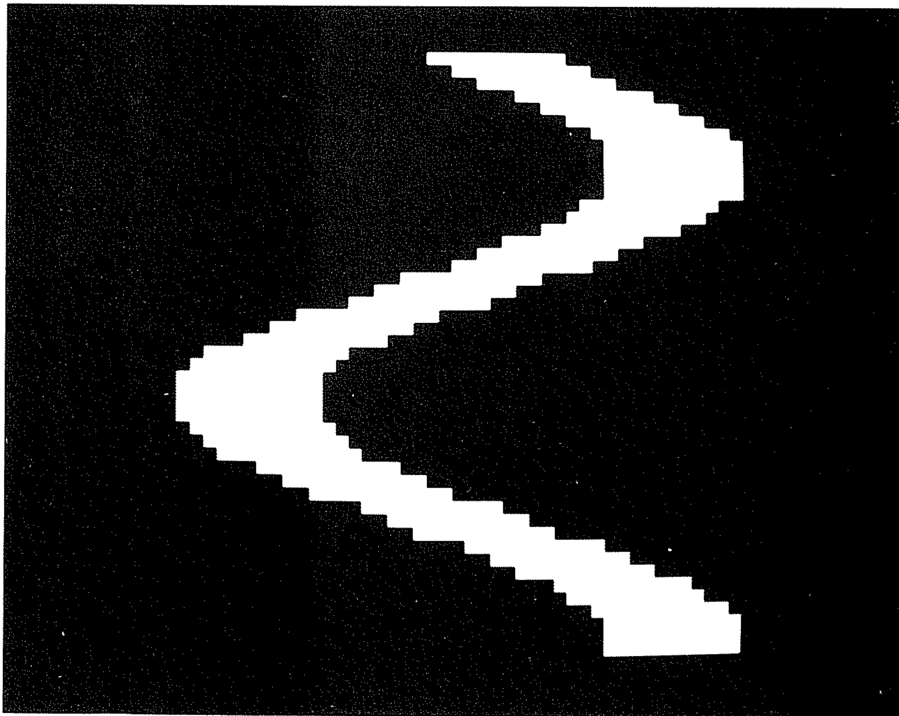


Photo 9-L-2

represents the least significant of the screen position, and L represents the pixel row to be set (1, 2, or 3). This loop draws a line in each of the pixel rows on the screen, with the drawing done by the USR call (line 270).

After the lines are drawn, the screen is reversed and then reversed back to its original several times (lines 310 to 340).

Chapter Summary

The video display of the TRS-80 Model I, III, or 4 can be scrolled left in machine language by repeatedly drawing a blank line down the left side of the screen, then performing a block move to the left on the entire screen contents.

A design can be created by loading the video memory with specified graphics characters in a pattern.

The screen can be split and moved to the sides horizontally by drawing a blank line down the center, then performing block moves on the left and right sides of each screen line.

The graphics characters can be reversed by subtracting the value of each graphics character on the screen from a certain number, then loading the difference back into the video memory.

Lines can be drawn on the screen using the assembly language SET instruction.

Meeting the Chapter Objectives

Here are the objectives for your review. Can you meet all the objectives?

A. Write an explanation of how an assembly language program that moves the screen sideways works.
B. Write an explanation of how an assembly language program that creates an alternate-character design on the screen works.
C. Write an explanation of how an assembly language program that splits the screen works.
D. Write an explanation of how an assembly language program that reverses the screen works.
E. Write an comparison between the BASIC SET instruction and the assembly language SET instruction.
F. Write an explanation of how an assembly language program that draws horizontal lines works.

More Programming Practice

1. Write an assembly language program that scrolls only characters on the top half of the screen to the left.

2. Write an assembly language program that makes a design with characters 141D and 189D.

3. Write an assembly language program that splits the screen more slowly. (Hint: use the DELAY ROM call.)

4. Write an assembly language routine that reverses the screen, then reverses it back to its original state.

5. Using the machine language line-drawing routine in a BASIC program, write a program that SETs every pixel on the video display.

Chapter Checkup

1. How does the left-scroll program in Listing 9-D solve the problem of the left-most row wrapping around to the right side during the block move?

2. What instruction in Listing 9-D moves the block of characters that scrolls the screen to the left?

3. Examine the program in Listing 9-E. What appears on the screen when lines 150 to 170 are executing?

4. What ROM routines are called by the program in Listing 9-F? What do they do? What assembly language instruction is used to call the routines?

5. In the program in Listing 9-G, which RETurn in fact returns from this whole program to whatever called it?

6. In the program in Listing 9-H, what must the user do to exit the BASIC program?

7. Consider the program in Listing 9-I. What character would be the reverse of graphics character 145?

8. What is the syntax of the SET instruction in BASIC? In assembly language?

9. What does the assembly language BIT instruction do?

10. In the BASIC program in Listing 9-L, why was it necessary to clear the screen with a graphics space (character 128) before drawing the lines?

Chapter 10
Assembly Language File Handling
In Graphics Progamming

OBJECTIVES:

At the end of this chapter, the reader will be able to perform the following tasks:

A. Write an explanation of why a programmer would want to use file handling to program graphics.
B. Write an explanation of the function, location, and conditions under which each of the following DOS calls are used: OPEN, POSN, READ, WRITE, and CLOSE.
C. Write a description of and the function of the Device Control Block (DCB).
D. Write an explanation of how a program can read data from the video memory into a data file.
E. Write an explanation of how a program can read data from a data file into the video memory.

INTRODUCTION

Chapters 9 and 10 discussed how to present graphics on the TRS-80 video display and how to manipulate them. This chapter shows how to use data files in assembly language for storage and later retrieval of graphics and text data to and from disk data files.

An obvious question is, "Why do you need to use disk data files to create screen graphics?" When an elaborate graphic is to be displayed on the screen, it may have a lot of variety on the screen. Many different graphics characters may be used to create a picture or other design. Unless there is a definite pattern to the graphic, programming must be in BASIC or in complex machine language routines which are beyond the scope of this text.

Each screen has 1024 characters on it, so it may take 1 K of memory to store it (unless the screens are generated). If the program had several screens, it could take several K of memory to store the graphics. And creating rapidly-changing screens (to effect limited animation) may become very difficult.

By storing a screen directly onto a disk file, the programmer eliminates the necessity of storing all the data for that screen in the program itself. And the same routines may be used to create, store, and display a wide variety of graphics screens.

The programs presented in this chapter allow the user to do the following:

1. Create a graphics screen with a screen editor
2. Save it at a specified point within a data file.
3. Recall the same graphics screen from the data file later.
4. Load several of these screens into the memory of the computer at once, and display them in rapid succession

without accessing the disk.

This process has several obvious advantages:

1. It allows the user to easily design the screens.
2. It allows the user to design similar screens which may be used in succession to create simple animation.
3. It avoids the use of data statements and other numeric representation of the graphics data.

It also has some disadvantages:

1. It is rather memory intensive. Each screen in memory requires 1 K of memory.
2. The animation is limited to the screens that can be stored in memory.
3. The initial load of the data from disk takes some time.
4. **THIS PROGRAM REQUIRES A 48K OR 64K TRS-80 MODEL III OR MODEL 4. The program requires Model III TRSDOS 1.3 (the current version of TRSDOS).**

ABOUT ASSEMBLY LANGUAGE FILE HANDLING

To this point, we have not discussed disk file handling. File handling in BASIC offers some of the same capabilities as file handling in machine language, but not nearly as much speed. When retrieving data from the disk to be displayed as graphics on the screen, the speed can become very important.

The file handling mentioned in this chapter will be done using Radio Shack's Disk Operating System, TRSDOS. A copy of the TRSDOS operating system comes with every disk TRS-80 computer when it is purchased. TRSDOS lets the assembly language programmer open disk files and read and write to them with relative ease.

TRSDOS has several routines in memory called DOS calls. These routines may be called from BASIC or machine language to perform certain tasks. The DOS calls used in the programs in this chapter use the DOS calls which manipulate data files. Many of these routines require the programmer to set aside an area of memory called a "Device Control Block" (DCB). Before the file is opened, the programmer places the filename of the file to be opened into the DCB. While the file is open, TRSDOS uses the DCB to store other information (which the programmer does not need to worry about). When the program closes the file, TRSDOS automatically puts the filename back into the DCB.

DOS CALLS USED IN THIS PROGRAM

OPEN. This routine (located at 4424H) opens a file which already exists on the disk. The programmer must put the file specification in the DCB before the file is opened. Here is what the programmer must set

before calling **OPEN:**

HL must point to a 256 byte area called the "buffer." This area must not be altered by the program while the file is open.

DE must point to the DCB.

B must contain the logical record length (LRL). For the progams in this chapter, the LRL will be 256, specified by using the number 0.

NOTE: The pointers and other file-handling manipulators in this chapter are based on an LRL of 256, the most commonly used LRL. When any other LRL is used, these pointers must be set up in a slightly different way. See your Disk System Owner's Manual for details.

POSN. This routine (located at 4442H) positions the disk read-write head at a specified point in the open file. Here is what the programmer must set before calling **POSN:**

DE must point to the DCB.

BC must contain the desired logical record number.

READ. This routine (located at 4436) reads the data from the current record into the buffer specified when the file was OPENed. After the record is read, the record number pointer is automatically incremented to be ready to read the next record. Here is what the programmer must set before calling **READ:**

DE must point to the DCB.

WRITE. This routine (located at 4439H) transfers the information from the buffer (specified when the file was opened) to the current record on the disk. After the write, the current record number is automatically incremented. Here is what the programmer must set up before calling **WRITE:**

DE must point to the DCB.

CLOSE. This routine (located at 4428H) closes the file. This must be done after file processing is done so the buffer will be emptied to the file and the directory entry will be updated. Here is what the programmer must set up before calling **CLOSE:**

DE must point to the DCB.

ABOUT THE PROGRAM

This chapter contains one long assembly language program and two BASIC programs that call it. The assembly language program has two main parts:

Lines 130 to 480 write the contents of the screen (the video memory) directly to the disk file as specified by the calling BASIC program.

Lines 490 to 840 read the contents of the disk file specified by the calling BASIC program and move them directly to the screen.

```
                100          ;FILE TO SCREEN PROGRAM
                110          ;By Dennis Tanner
D6D8            120          ORG  55000D
D6D8 CD7F0A     130 START:   CALL GETARG
D6DB E5         140          PUSH HL           ;STORE PTR FOR REC#
D6DC 23         150          INC  HL           ;PT HL TO FILENAME
D6DD 54         160          LD   D,H          ;POINTING TO DCB
D6DE 5D         170          LD   E,L          ;POINTING TO DCB
D6DF 21C0DA     180          LD   HL,BUFFER    ;SET BUFFER
D6E2 0600       190          LD   B,00H        ;LRL
D6E4 CD2444     200          CALL OPEN         ;OPEN FILE
D6E7 C1         210          POP  BC           ;GET REC #
D6E8 BA         220          LD   A,(BC)       ;REC # INTO A
D6E9 0600       230          LD   B,00H        ;ZERO MSB OF REC #
D6EB 4F         240          LD   C,A          ;REC # INTO C
D6EC CD4244     250          CALL POSN         ;POSITION HEAD
D6EF 215ED7     260          LD   HL,PS        ;PS (RAM POS) TO HL
D6F2 363B       270          LD   (HL),3BH     ;MSB OF VIDRAM TO PS-1
D6F4 23         280          INC  HL           ;TO STORE LSB
D6F5 3600       290          LD   (HL),00H     ;LSB OF VIDRAM TO PS+1
D6F7 215ED7     300 WRLOOP:  LD   HL,PS        ;COUNTER TO HL
D6FA 7E         310          LD   A,(HL)       ;MSB TO A
D6FB 3C         320          INC  A            ;A=A+1; PS=PS+256
D6FC 77         330          LD   (HL),A       ;BACK TO PS
D6FD FE40       340          CP   40H          ;BEYOND VIDRAM?
D6FF 2817       350          JR   Z,CLOS       ;IF SO, TO CLOS
D701 D5         360          PUSH DE
D702 3A5ED7     370          LD   A,(PS)       ;COUNTER TO A
D705 67         380          LD   H,A          ;MSB OF CURNT VR TO H
D706 3A5FD7     390          LD   A,(PS+1)     ;LSB OF COUNTER
D709 6F         400          LD   L,A          ;LSB OF CURNT VR TO L
D70A 11C0DA     410          LD   DE,BUFFER    ;TO BUFFER
D70D 010001     420          LD   BC,100H      ;COUNTER
D710 EDB0       430          LDIR              ;MOVE
D712 D1         440          POP  DE
D713 CD3944     450          CALL WRITE        ;FIRST 1/4 VID TO DISK
D716 18DF       460          JR   WRLOOP       ;ONE MORE TIME
D718 CD2844     470 CLOS:    CALL CLOSE        ;CLOSE FILE
D71B C9         480          RET               ;BYE!
D71C CD7F0A     490          CALL GETARG       ;GET REC# & FILENAME
D71F E5         500          PUSH HL           ;STORE PTR FOR REC#
D720 23         510          INC  HL           ;PT HL TO FILENAME
D721 54         520          LD   D,H          ;POINT DE TO FILENAME
D722 5D         530          LD   E,L
D723 21C0DA     540          LD   HL,BUFFER    ;SET BUFFER
D726 0600       550          LD   B,00H        ;LRL
D728 CD2444     560          CALL OPEN         ;OPEN FILE
D72B C1         570          POP  BC           ;GET REC#
D72C 0A         580          LD   A,(BC)       ;REC# INTO A
D72D 0600       590          LD   B,00H        ;LRL
```

```
D72F 4F      600         LD    C,A          ;REC# INTO C
D730 CD4244  610         CALL  POSN         ;POSITION HEAD
D733 215ED7  620         LD    HL,PS        ;PS (RAM POS) TO HL
D736 363B    630         LD    (HL),3BH     ;MSB OF VIDRAM -1
D738 23      640         INC   HL           ;TO STORE LSB
D739 3600    650         LD    (HL),00H     ;LSB OF VIDRAM TO PS+1
D73B 215ED7  660 RELOOP: LD    HL,PS        ;POS COUNTER TO HL
D73E 7E      670         LD    A,(HL)       ;MSB TO A
D73F 3C      680         INC   A            ;A=A+1; PS=PS+256
D740 77      690         LD    (HL),A       ;BACK TO PS
D741 FE40    700         CP    40H          ;BEYOND VIDRAM?
D743 28D3    710         JR    Z,CLOS       ;IF SO, TO CLOS
D745 CD3644  720         CALL  READ         ;1ST 1/4 VID FROM DISK
D748 D5      730         PUSH  DE
D749 3A5ED7  740         LD    A,(PS)       ;COUNTER TO A
D74C 67      750         LD    H,A          ;MSB OF CURNT VR TO H
D74D 3A5FD7  760         LD    A,(PS+1)     ;LSB OF COUNTER
D750 6F      770         LD    L,A          ;LSB OF CURNT VR TO L
D751 E5      780         PUSH  HL           ;SAVE VIDRAM
D752 D1      790         POP   DE           ;GET VIDRAM
D753 21C0DA  800         LD    HL,BUFFER    ;FROM BUFFER
D756 010001  810         LD    BC,100H      ;COUNTER
D759 EDB0    820         LDIR               ;MOVE
D75B D1      830         POP   DE
D75C 18DD    840         JR    RELOOP       ;BACK TO READ LOOP
0A7F         850 GETARG  EQU   0A7FH
4424         860 OPEN    EQU   4424H
4428         870 CLOSE   EQU   4428H
4442         880 POSN    EQU   4442H
4436         890 READ    EQU   4436H
4439         900 WRITE   EQU   4439H
DAC0         910 BUFFER  EQU   56000D
D75E         920 PS      DEFW  0000H
D6DB         930         END   START
```

Listing 10-A

The BASIC program that calls the machine language program in Listing 10-A sets up a 33-character string that contains the following:

1) A one-byte character representing the first record number of the desired picture in the file. Each file can contain up to 255 pictures. This character tells which record in the file to address for the desired picture. If the program is addressing the fourth record in the file, the first character in the string in the BASIC program will be CHR$(4).

2) The file specification. This would be the filename, extension, password (if applicable), and drive specification.

3) Spaces to fill out the rest of the 33 characters.

The BASIC program then assigns that 33-character string to a string variable and retrieves the VARPTR (variable pointer) of the string. In the first BASIC program, for example, the 33-character string is named C$. V is the VARPTR of C$. When the machine language program is called, this format is used:

X=USR1(V) or X=USR2(V).

The USR call can be addressed in this format **to pass a value from the BASIC program to the machine language program.** The value we are passing is the number that points to the 33-character string in memory. **This string will contain the DCB (device control block) described above.**

When a value is passed from a BASIC program to a machine language program, the machine language must take a special step before doing anything else. It must call the ROM routine called "GETARG" at location 0A7FH. This routine retrieves the value that was passed from the BASIC program in the USR call and places it in HL. The program in Listing 10-A defines the GETARG label in line 850 and calls it first thing, in line 130.

At that point, HL contains the pointer to the 33-character string. Line 140 PUSHes HL onto the stack to store the record number of the desired picture for later use. Line 150 INCrements HL, and then HL points to the 32-character area containing the file specification and spaces. In other words, HL points to the DCB.

Here is a line-by-line description of the rest of the first routine:

Line	Description
160	This line and line 170 load the address pointing to the DCB from HL to DE where it will be stored. Remember that the DE must point to the DCB before the OPEN routine is called.
170	Described above.
180	Line 910 defines BUFFER as a pointer to the address 56000D. This will be a pointer to the 256-byte block of memory required for opening the file. Line 180 points HL to the buffer, as must be done before OPEN can be called.
190	B now contains 0, the logical record length (LRL). Remember that to specify an LRL of 256, B must be loaded with 0.
200	Opens the file.
210	Retrieves the pointer to the record number of the desired picture and stores it in BC.
220	Gets the record number into A, for later transfer to C.
230	Loads zero into B. This assures that BC will contain only the value contained in C.
240	Loads the record number into C. Now BC contains the record number.
250	Since BC contains the record number and DE still points to the DCB (see lines 160 and 170), the program is ready to call the POSN DOS call to position the disk drive head to the right record. This

line calls POSN.

260 Loads HL with the PS (screen position) pointer, as defined in line 880.

270 Loads 3BH into the first two bytes of PS.

280 Increments HL to point to the next two bytes of PS.

290 Loads the last two bytes of PS with 0. Now PS contains 3B00H which is 256 bytes before the beginning of video memory (3C00H). When the MSB (first two bytes) of PS is incremented in line 320, PS will point to 3C00H.

300 This line begins the WRLOOP which reads the video memory and stores it on the disk file. The screen has 1024 bytes, and each record has 256 bytes, so it takes exactly 4 records to hold one screenful. This line loads the current screen position (PS, initially 3B00H as mentioned above) into HL.

310 The MSB of PS is loaded into A.

320 A is incremented. Since this is the MSB of PS, PS will be increased by 100H (256D) when this value is loaded back to PS in the next line.

330 A is loaded back into the MSB of PS.

340 A is compared with 40H, the MSB of the position <u>past</u> the video memory.

350 If the compare returns zero, and A is therefore 40H, the program jumps down to CLOS (line 470) to close the file and return to the BASIC program.

360 The value in DE, which now points to the DCB, is PUSHed onto the stack for later use.

370 The next few lines set up the LDIR move of the graphics from the video memory to the buffer. The MSB of PS (the screen position) is loaded into A.

380 The MSB of PS is loaded into H.

390 The LSB of PS is loaded into A.

400 The LSB of PS is loaded into L. This finishes setting up the "move from" area (part of the video memory) by loading it into HL.

410 DE is loaded with the BUFFER pointer. This is the "move to" area.

420 BC is loaded with 100H (which is 256D) as the counter of the LDIR coming up.

430 The data is moved from the video memory to the buffer.

440 The top number from the stack is POPped to DE. Now DE once again points to the DCB.

450 Since DE points to the DCB, the WRITE routine can be called. This line calls the WRITE routine.

460 The routine jumps back to WRLOOP for the next record.

470 This line (labelled CLOS) calls the CLOSE routine to close the file.

480 Returns to the BASIC program.

Thus the first routine reads data from the video memory and transfers it to the buffer in 256 byte segments. The disk routines then move the data from the buffer to the record of the file. Since the WRITE routine automatically moves the record pointer to the next record after the write, it is set up for the next area of the video memory.

The second major part of this program does just the reverse. It reads each record into the buffer and transfers the data from the buffer directly to the video memory. The details of the routine are similar to the details of the first routine, and they are left to the reader.

THE BASIC PROGRAM THAT CALLS THIS ROUTINE

Since this program is specifically written to be called by a BASIC program, we must write a BASIC program to call it. The program in Listing 10-B does just that.

```
10 'Program to interface with machine language file-handling
      routines. Protect memory at 48000D.
20 CLEAR 500: GOTO 110
30 FOR J=0 TO LEN(A$)/2-1
40  TE$ = MID$(A$,J*2+1,2)
50  MSB=ASC(LEFT$(TE$,1)): MSB=MSB-48:
     IF MSB>9 THEN MSB=MSB-7
60  LSB=ASC(RIGHT$(TE$,1)): LSB=LSB-48:
     IF LSB>9 THEN LSB=LSB-7
70  NU = MSB*16+LSB
80  POKE BG+J, NU
90 NEXT J
100 RETURN
110 A$="CD7F0AE523545D21C0DA0600CD2444C10A06004FCD424
     4215ED7363B233600215ED77E3C77FE402817D53A5ED7673A5
     FD76F11C0DA010001EDB0D1CD394418DFCD2844C9"
      'Screen-to-disk routine
120 BG = &HD6D8: GOSUB30: DEFUSR1 = BG
130 A$="CD7F0AE523545D21C0DA0600CD2444C10A06004FCD424
     4215ED7363B233600215ED77E3C77FE4028D3CD3644D53A5ED
     7673A5FD76FE5D121C0DA010001EDB0D118DD0000"
     'Disk-to-screen routine
140 BG = &HD71C: GOSUB30: DEFUSR2 = BG
150 INPUT "1=VID TO DISK  2=DISK TO VID"; A
160 INPUT "FILENAME"; C$: D$=C$:
     INPUT "PICTURE#"; B: B=(B-1)*4:
     C$ = CHR$(B) + C$ + CHR$(13) + STRING$ (31-LEN(C$), 32)
170 V = VARPTR(C$): V = PEEK(V+1) + PEEK(V+2) * 256:
     IF V>(2[15) THEN V=V-(2 16)
180 IF A=2 THEN GOSUB240: X=USR2(V): GOTO220
190 CLS: PRINT "OPENING FILE...": OPEN"R",1,D$: CLOSE:
     XX=RND(64)+128: FOR J=15360 TO 16383: POKE J,XX: NEXT J
200 PRINT@0, "STARTING W/RECORD#" B;
210 X = USR1(V)
220 I$ = INKEY$
230 I$ = INKEY$: IF I$="" THEN 230 ELSE CLS: GOTO 150
```

```
240 ONERRORGOTO 260
250 OPEN"I",1,D$: CLOSE: GOTO 270
260 PRINT "FILE DOESN'T EXIST": ONERRORGOTO0: END
270 ONERRORGOTO 0: RETURN
```

Listing 10-B

Lines 30 to 100 of this program contain the routine to decode the hex versions of the machine language programs and POKE them into memory.

Lines 110 to 140 contain the hex data for the two parts of the program in Listing 10-A. They call the POKEing subroutine and define the USR calls to be used.

Line 150 presents a one-line menu. The program lets you choose to load the information from the screen to the disk, or load the information from the disk to the screen.

Line 160 accepts the filename (C$) and picture number (B). It calculates the record number to be specified according to the picture number. It then reassigns C$ with the record number (CHR$(B)), the filename (C$), a carriage return to terminate the filename (CHR$(13)), and a string of spaces to pad the DCB.

Line 170 finds the variable pointer of the string, and from the VARPR finds the location of the string data in memory. The number pointing to that data is reassigned to V.

Line 180 executes the "Disk to vid" instructions if the user has chosen option 2. It checks for the existence of the file (GOSUB 240). If the file exists, it calls the "Disk-to-screen" program (USR2(V)). By specifying V in the USR call, the program passes the pointepress.

Line 190 is executed if the user has chosen option 1. It opens the file and closes it first thing, since the machine language OPEN routine works only on files that already exist. It then chooses a graphics character and puts it in every position of the screen.

Line 200 displays a message telling which record of the file will be the first one used for this picture. This helps when pulling the picture out of the file to be sure the proper picture was accessed.

Line 210 calls the machine language program that reads the data from the screen and writes it to the file.

Lines 210 to 230 await a keypress and then return to the options.

Lines 240 to 270 contain the error-trapping routine that makes sure the file exists if it is to be read by the machine language program. OPENing a file for input ("I") will return an error message when the file

does not exist on the disk. ?question: In the to be read by the machine language program. OPENing a file for input ("I") will return an error message when the file does not exist on the disk. ?question: In the program in Listing 10-B, what will happen in lines 240 to 270 if the file does not exist?

This program can be used on any file with a logical record length of 256. If you have a data file from another program, you may use option 2 to move its contents to the display for a good look at them.

Although this BASIC program shows the capabilities of the machine language subroutine, it does not allow the user to create his or her own graphic screens and to save them on the disk. The program that follows does that and more.

SAVING AND RETRIEVING SCREENS OF YOUR OWN DESIGN

The BASIC program that follows culminates this book by offering many of the features and using many of the concepts from different sections of the book. It has these features:

+ A screen editor allows you to create unique screens. Besides the regular SET/RESET instructions found in our previous screen editors, this one draws lines and circles, clears the screen with a graphics character (191), moves the cursor to a specified area, and allows alphanumeric input at the keyboard.

+ The screen can be saved at any time. If you desire, the screen can be recalled, edited and saved again under the same or a different filename and position in the file. This permits you to make screens with most of the positions the same, but a few differences. Such screens can be animated as described next.

+ You may specify up to eight pictures to loaded into memory. These are loaded into adjacent sections of high memory one after another. They are then brought back down to the video memory one at a time with slight pauses, but quickly enough to appear to be animated.

As mentioned in the introduction to this chapter, this process is rather memory intensive, but it can be used to create some dramatic graphic effects.

```
1 ' COMBINED PROGRAM FOR CHAPTER 10 By Dennis Tanner
      Protect memory at 48000D.
5 GOTO 20
6 LO = PEEK(16416) + PEEK(16417) * 256: L1 = LO: I$ = INKEY$:
      S$ = "": CU = 32
7 CU = 168 - CU: POKE LO, CU: FOR JJ = 1 TO 20: I$ = INKEY$:
      IF I$ = "" THEN NEXT JJ: GOTO7
      ELSE JJ = 20: NEXTJJ: I = ASC(I$):
      IF I = 8 AND S$ <> "" THEN POKE LO, 32: LO = LO - 1:
```

```
     S$ = LEFT$(S$,LEN(S$)-1): GOTO7
8 IF I = 13 AND S$ <> "" THEN POKE LO, 32: RETURN
     ELSE IF I < LV OR I > HV THEN 7
     ELSE IF LO - L1 = LE THEN 7
     ELSE POKE LO, I: S$ = S$ + I$: LO = LO + 1: CU = 32: GOTO7
20 '
     MAIN BODY OF PROGRAM

30 CLEAR 500                'INITIALIZE VARIABLES
40 GOSUB 200                'PRESENT TITLE SCREEN
50 GOSUB 500                'LOAD IN MACHINE LANGUAGE
                             PROGRAMS
60 GOSUB 800                'PRESENT OPTION LIST
70 ON CH GOTO 80, 90, 100   'BRANCH ACCORDING TO CHOICE
80 GOSUB 900                'CREATE A NEW FILE
85 GOTO 60
90 GOSUB 1000               'EDIT AN EXISTING FILE
100 GOSUB 1500              'VIEW THE SCREENS
200 '
     Print Title Screen

210 CLS
220 PRINT@ 88, STRING$(16,140)
230 PRINT@ 152, "Graphics Program"
240 PRINT@ 216, STRING$(16,140)
250 PRINT@ 980, "< < < Please Wait > > >";
260 RETURN
299 END
300 '
     POKE Machine Code to Memory

310 FOR J=0 TO LEN(A$)/2-1
320  TE$ = MID$(A$,J*2+1,2)
330  MSB = ASC(LEFT$(TE$,1)): MSB = MSB - 48:
     IF MSB > 9 THEN MSB = MSB - 7
340  LSB = ASC(RIGHT$(TE$,1)): LSB = LSB - 48:
     IF LSB > 9 THEN LSB = LSB -7
350  NU = MSB * 16 + LSB
360  POKE BG +J, NU: POKE 15423, 74 - PEEK(15423)
370 NEXT J
380 RETURN
400 '
     Routine That Moves Screens with LDIR

410 POKE &HBC00, INT(MT/256): POKE &HBC01, MT-INT(MT/256)*256
420 POKE &HBC02, INT(HM/256): POKE &HBC03, HM-INT(HM/256)*256
430 POKE &HBC04, INT(MF/256): POKE &HBC05, MF-INT(MF/256)*256
440 X = USR1(0)
450 RETURN
500 '
     Machine Language Programs to be POKEd into Memory
```

```
510 A$="2100BC562101BC5E2102BC462103BC4E3A04BC673A05B
     C6FEDB0C9" 'LDIR PROGRAM
520 BG = 48000 - 2 16
530 DEFUSR1 = BG
540 GOSUB 310
550 A$="21FF3B237CFE40C836BF18F7" 'SCREEN FILL
560 BG = 49000 - 2 16
570 DEFUSR2 = 49000 - 2 16
580 GOSUB 310
590 A$="002188C336002188C37EFE4028293C77CD65C318F100014
     00021003C7CFE4028053620091 8F621FF3F362011003C21013C
     01FF03EDB0C9C90000" 'MOVE SCREEN
600 BG = 50000 - 2 16
610 DEFUSR3 = 50000 - 2 16
620 GOSUB 310
630 A$="0021FE3B16BF1E8CCD67C7010080CD6000CD8D02C279C
     721FE3B168C1EBFCD67C7010080CD6000CD8D02C279C718D1
     23237CFE40280BE5DDE1DD7200DD730118EEC9" 'DESIGN
640 BG = 51000 - 2 16
650 DEFUSR4 = BG
660 GOSUB 310
670 A$="CD27CBCD4BCBC92185CB36E023363B3A86CB673A85CB
     6F014000097CFE40C836207D3285CB7C3286CB18E42183CB36
     212183CB7E3D77FE00C8CD5ECB18F221FF3B010100097CFE4
     0C8E5D109012000EDB0011E0009E5D12B012000EDB801210009
     18DE00000000" 'SPLIT
680 BG = 52000 - 2 16
690 DEFUSR5 = BG
700 GOSUB 310
710 A$="002157D37E572154D37E4F2155D37E4760693E005F6069C
     B7E20033E80772156D37EFE012809FE022810FE032817C9CD4B
     D3CBFECBC6CBCE1814CD4BD3CBFECBD6CBDE1809CD4BD3
     CBFECBE6CBEE7B3C5F23444DF5BA2806F118BA6069C9F1C9"
     'LINE DRAWER
720 BG = 54000 - 2 16: GOSUB310: DEFUSR6 = 54000 - 2 16
730 A$="0021FF3B237CFE40280D7EFE7F38F57E473E3F907718EDC9"
     'REVERSE ROUTINE
740 BG = 53000 - 2 16: GOSUB 310: DEFUSR7 = 53000 - 2 16
750 A$="CD7F0AE523545D21C0DA0600CD2444C10A06004FCD424
     4215ED7363B233600215ED77E3C77FE402817D53A5ED7673A5
     FD76F11C0DA010001EDB0D1CD394418DFCD2844C9" 'WRITE FILE
760 BG = &HD6D8: GOSUB 310: DEFUSR8 = BG
770 A$="CD7F0AE523545D21C0DA0600CD2444C10A06004FCD424
     4215ED7363B233600215ED77E3C77FE4028D3CD3644D53A5ED
     7673A5FD76FE5D121C0DA010001EDB0D118DD0000" 'READ FILE
780 BG = &HD71C: GOSUB 310: DEFUSR9 = BG
790 RETURN
800 '
     Present Option List
```

```
810 CLS: PRINT@ 343,CHR$(31); "1  Create a file"
820 PRINT@ 407, "2  Edit a file"
830 PRINT@ 471, "3  View screens"
840 PRINT@ 599, "Enter selection: ";
850 LV=49: HV=51: LE=1: GOSUB6
860 CH = VAL(S$)
870 RETURN
900 '
     Create a New File

910 PRINT@ 727, "Filename of new file? ";
920 LV=47: HV=90: LE=10: GOSUB 6: F$=S$
930 ON ERROR GOTO 960
940 OPEN"I",1,F$: CLOSE
950 PRINT @791, "The file already exists.";: FOR KK=1TO300: NEXTKK:
     GOTO 970
960 OPEN"R",1,F$: CLOSE: RESUME 970
970 ONERRORGOTO0: RETURN
1000 '
Edit an Existing File

1010 PRINT@ 727, CHR$(31);"Filename? "; CHR$(31);
1020 LV=48: HV=92: LE=10: GOSUB6: F$=S$
1030 OPEN"R",1,F$: CLOSE
1040 PRINT@ 791, "Screen number? "; CHR$(31);
1050 LV=48: HV=57: LE=2: GOSUB 6: SN = VAL(S$): IF SN>20 THEN 1040
1060 PRINT@ 855, "Load existing screen? "; CHR$(30);
1061 LV=65: HV=90: LE=1: GOSUB6: YN$=S$:
     IF YN$ <> "N" AND YN$ <> "Y" THEN 1060
1062 IF YN$ <> "Y" THEN 1095
     ELSE PRINT@919, "Save under what name? "; CHR$(30);
1064 LV=48: HV=92: LE=10: GOSUB6: NF$=S$
1066 PRINT@ 983, "Screen number? "; CHR$(30);
1068 LV=48: HV=57: LE=2: GOSUB6: NN=VAL(S$): IF NN > 20 THEN 1066
1080 C$=F$: D$=C$: B=SN: B=(B-1)*4:
     C$=CHR$(B)+C$+CHR$(13)+STRING$(31-LEN(C$),32):
     V=VARPTR(C$): V=PEEK(V+1)+PEEK(V+2)*256:
     IF V>(2 15) THEN V = V-(2 16)
1090 X = USR9(V): GOTO 1100
1095 NF$=F$: NN=SN: CLS
1100 ONERRORGOTO 1310
1120 I$=INKEY$: X=0: Y=0
1130 PRINT@ 960, X; Y; CHR$(30);
1140 I$=INKEY$: IF I$="" THEN POKE16383,74-PEEK(16383): GOTO 1140
1150 IF I$=CHR$(9) THEN X=X+1: SET(X,Y): GOTO1130
1160 IF I$=CHR$(8) THEN X=X-1: SET(X,Y): GOTO1130
1170 IF I$=CHR$(91) THEN Y=Y-1: SET(X,Y): GOTO1130
1180 IF I$=CHR$(10) THEN Y=Y+1: SET(X,Y): GOTO1130
1190 IF I$=CHR$(32) THEN RESET(X,Y): GOTO1130
1200 IF I$="Y" OR I$="y" THEN PRINT@960, CHR$(30);: GOTO 1350
1210 IF I$="C" OR I$="c" THEN PRINT@975, "Radius? ";: LV=48: HV=57:
```

```
      LE=2: GOSUB6: RA=VAL(S$):
      FOR AN=0 TO 6.3 STEP .02:
            X1 = X+COS(AN)*RA: Y1 = Y+SIN(AN)*RA*.55: SET(X1,Y1):
      NEXT AN:
      PRINT@ 960, CHR$(30);: GOTO 1130
1220 IF I$ = "M" OR I$="m" THEN PRINT@ 975, CHR$(30);
      "Move to:  X? ";: LV=48: HV=57: LE=3: GOSUB6: XM=VAL(S$):
      PRINT@ 998, "Y? ";: LE=2: GOSUB6: YM=VAL(S$): X=XM: Y=YM:
      PRINT@ 960,CHR$(30);: GOTO 1130
1230 IF I$=CHR$(31) THEN CLS: GOTO1130
1240 IF I$<>"L" AND I$<>"l" THEN 1260
      ELSE PRINT@ 975, "Other endpoint: X? ";: LV=47: HV=58: LE=3:
      GOSUB6: EX=VAL(S$): PRINT@ 1000, "Y? ";: LE=2: GOSUB6:
      EY=VAL(S$): SX=X: SY=Y: LE=SQR((EX-SX) 2 +(EY-SY) 2)
1250 FOR J=0 TO LE: X=SX+(EX-SX)*J/LE: Y=SY+(EY-SY)*J/LE:
      SET(X,Y): NEXTJ: X=INT(X): Y=INT(Y): GOTO1130
1260 IF I$="R" OR I$="r" THEN X1=USR7(0): GOTO1130
1270 IF I$="F" OR I$="f" THEN X1=USR2(0): GOTO1130
1280 IF I$="A" OR I$="a" THEN PRINT@ 960,CHR$(30);:
      PRINT@ INT((Y)/3)*64+INT(X/2),"";: LV=32: HV=90: LE=64: GOSUB6:
      X = (LO-15360-INT((LO-15360)/64)*64)*2:
      Y = INT((LO-15360)/64)*3+1: GOTO 1130
1300 GOTO 1130
1310 PRINT@ 960, "ERROR...";
1320 FOR K=1 TO 400: NEXT K
1330 PRINT@ 960, "      ";
1340 RESUME NEXT
1350 C$=NF$: D$=C$: B=NN: B=(B-1)*4:
      C$ = CHR$(B) +C$ +CHR$(13)+ STRING$(31-LEN(C$),32)
1360 V = VARPTR(C$): V = PEEK(V+1) + PEEK(V+2) * 256:
      IF V>(2 15) THEN V=V-(2 16)
1380 X = USR8(V)
1390 PRINT@ 960, NF$; "  PICTURE #"; NN;: I$=INKEY$
1400 I$=INKEY$: IF I$="" THEN 1400 ELSE CLS: GOTO 60
1410 ONERRORGOTO 1430
1420 OPEN"I",1,D$: CLOSE: GOTO 1440
1430 PRINT "FILE DOESN'T EXIST": ONERRORGOTO0: END
1440 ONERRORGOTO 0: RETURN
1500 '
      View the Screens

1510 CLS: PRINT@ 400,"View how many screens? "; CHR$(31);
1520 LV=47: HV=58: LE=1: GOSUB6: NS=VAL(S$): IF NS>8 THEN 1510
1530 FOR J=1 TO NS
1540  PRINT@ 400+J*64,"Filename? ";
1550  LV=47: HV=90: LE=10: GOSUB6: F$(J)=S$
1560  PRINT@ 425+J*64,"Screen #? "; CHR$(30);
1570  LV=48: HV=57: LE=2: GOSUB6: SN(J)=VAL(S$):
      IF SN(J)>20 THEN 1560
1580 NEXT J
1600 FOR J=1 TO NS
```

```
1610 C$=F$(J): B=SN(J): B=(B-1)*4:
     C$ = CHR$(B) + C$ + CHR$(13) + STRING$(31-LEN(C$),32):
     V=VARPTR(C$): V = PEEK(V+1) + PEEK(V+2) * 256:
     IF V>2 15 THEN V=V-2 16
1620 X=USR9(V): MF=15360: MT=55400+J*1100: HM=1024: GOSUB410
1630 NEXT J
1700 FOR J=1 TO NS
1720 MF=55400+J*1100: MT=15360: HM=1024: GOSUB410
1730  FOR KK=1 TO 10: NEXT KK
1740 NEXT J
1800 J=RND(3): ON J GOSUB 1810,1820,1830: CLS: GOTO 1700
1810 X1=USR3(0): RETURN
1820 X1=USR5(0): RETURN
1830 FOR K=1 TO 5: X1=USR7(0): FORKK=1TO50: NEXTKK: NEXTK:
     RETURN
```

Listing 10-C

When you run this program, the first screen will contain the title. It will remain for a minute or so while all the machine language routines are POKEd into memory with the routine in lines 300 to 380. A blinking asterisk has been added to line 360 so you will know that the program is indeed executing during the pause.

To use this program, first decide upon the filename you want to use for saving your graphics. (This file can contain several screen files, and you can edit them with relative ease.) If the file does not yet exist on the diskette, first initialize the file by choosing option 1, "Create a file." **Any file that is to be used with must program must either 1) already exist on the diskette, or 2) be initialized using option 1 of the program.**

To edit a file, choose option 2. The program will then prompt you for a filename and screen number within the file. It will ask if you are loading an existing screen, and if you are, it will ask you which filename and screen number to save it back out under after it is re-edited. (If you want to save it back to the same filename and screen number, specify the same filename and screen number when asked.)

The graphics screen will appear, and you can begin editing it. The X- and Y-coordinates appear on the bottom line of the display. You can use the arrow keys and the space bar as before. The blinking asterisk tells you that the program is awaiting an instruction from you.

While the asterisk is blinking, you have several other options, too.

+ You may press the F key to fill the entire graphics area with character 191, the completely filled graphics block.
+ You may press the R key to reverse all the graphics characters on the screen. For reasons detailed later, it is a good idea to press F and then R to clear the screen at the beginning of each graphics screen.

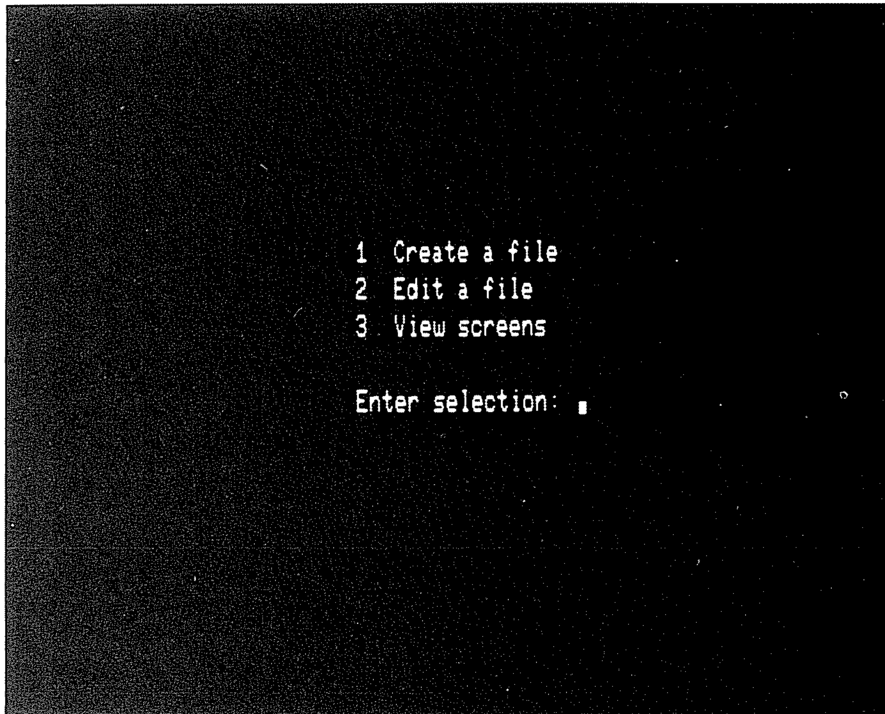


Photo 10-C-1

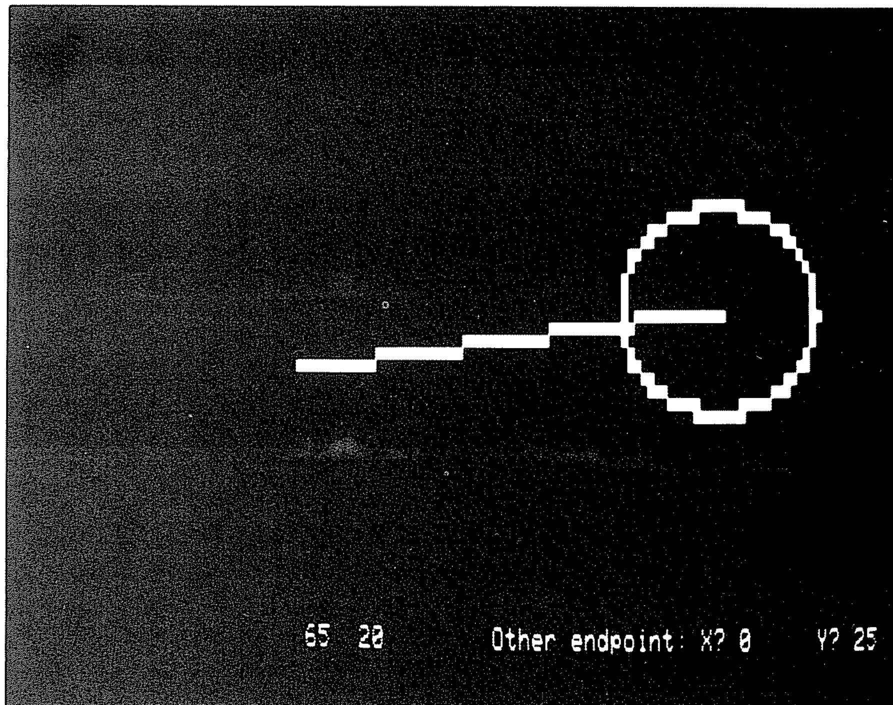


Photo 10-C-2

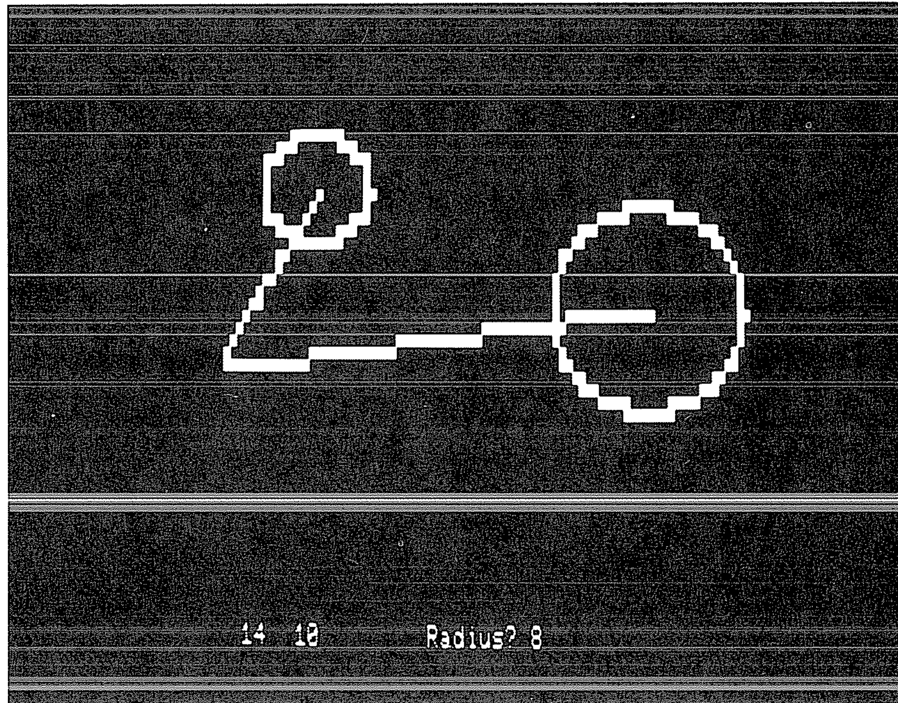


Photo 10-C-3

+ You may press the C key to draw a circle. The current cursor location will be used as the center of the circle. You will be prompted for the radius of the circle, and it will be drawn.
+ You may press the L key to draw a line. The current cursor location will be one end of the line, and the program will prompt you for the other endpoint.
+ You may press the M key to move the cursor without drawing any lines or points. The program will prompt you to specify the new point.
+ You may press the A key to type letters and numbers at the keyboard and have them appear on the screen. To return to the graphics editor, press ENTER.

For effective animation, you may want part of the screen to move and part of it to remain. To accomplish this, use the procedure shown in this example.

EXAMPLE USE OF THE PROGRAM

Choose a picture idea and a filename. For example, you may choose to draw a flower that "grows" on the screen. This flower will have seven different stages. The flower's filename will be F.

After you have typed in this program, save it on the diskette. RUN the program and choose option 1, "Create a file." In response to the "Filename of new file?" question, type F.

The option list will return to screen. Choose option 2, "Edit a file." The "Filename?" prompt will reappear. Type F. To the "Screen number?" prompt, type 1. To the "Load existing screen?" question, type N.

The screen editor will now appear with the asterisk blinking in the lower right corner and the "0 0" coordinates showing on the bottom line of the display. Type F the fill the screen and R to reverse the graphics from the screen. The reason for this is, later reversing of the screen will be effective after using this options, because the screen will be filled with a graphics space (character 128) rather than the regular space.

You may move the cursor to the center of the screen using the M instruction. You may use the arrow keys or the L instruction to draw the stem and leaves of the flower. You may want to use the C instruction to draw the blossom.

When the graphics has been completed, press Y to save the screen as screen 1 of the file called F.

To create the next screen, choose option 2 again. To the "Filename?" prompt, type F again. To the "Screen number?" prompt, type 1. To the "Load existing screen?" prompt, type Y. To the "Save under what name?" prompt, type F. To the "Screen number?" prompt, type 2.

The first screen you created will be displayed on the screen again, and you may edit it. You will probably want to change some parts of the graphic and leave some of the it the same. When you have finished, press Y, and your new graphic will be saved as screen 2 of the file F.

By repeating process from screen 2 to screen 3, screen 3 to screen 4, etc, you can create a set of screens which can be used for animation.

To view the screens, choose option 3, "View screens." It will ask you "View how many screens?" If you have created 7 successive graphic screens that shows the growth of a flower, type 7. For the number of screens that you have specified, the "Filename?" and "Screen #?" prompts will appear. In the sequence you want them to appear, type in the filenames and screen numbers you specified when the screens were created.

After you press enter for the last screen number, the program will load the screen files in one at a time. Each screen full of graphics will be sent to an area in high memory for later recall. After all the screens are loaded into memory, the program starts displaying them one by one, in sequence, fast enough so they appear animated. Since they are already loaded into memory, they can be displayed very quickly.

The details of the programming techniques are again left to the reader.

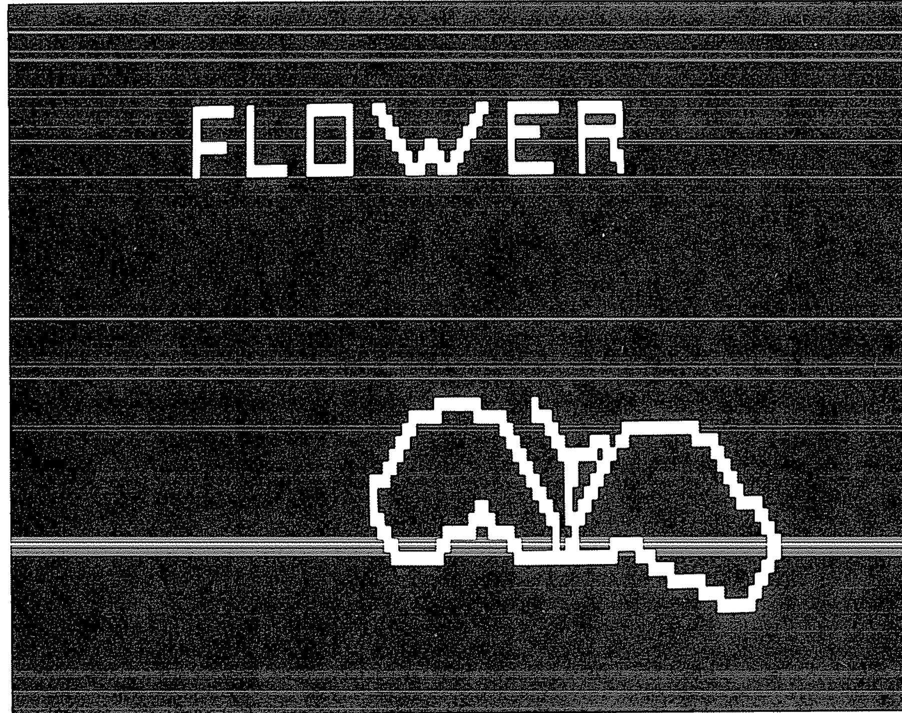


Photo 10-C-4

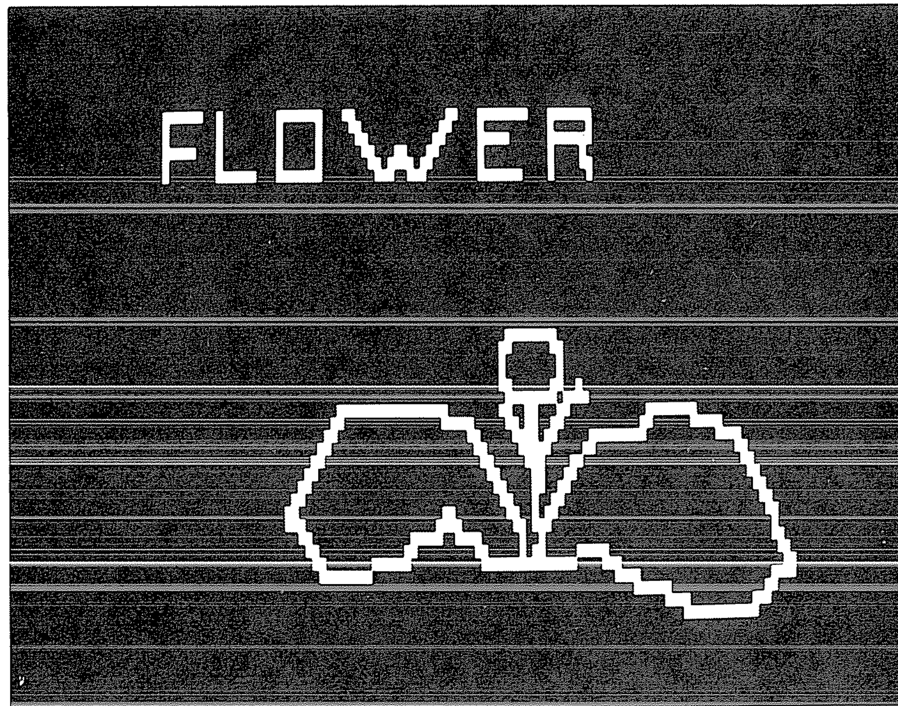


Photo 10-C-5

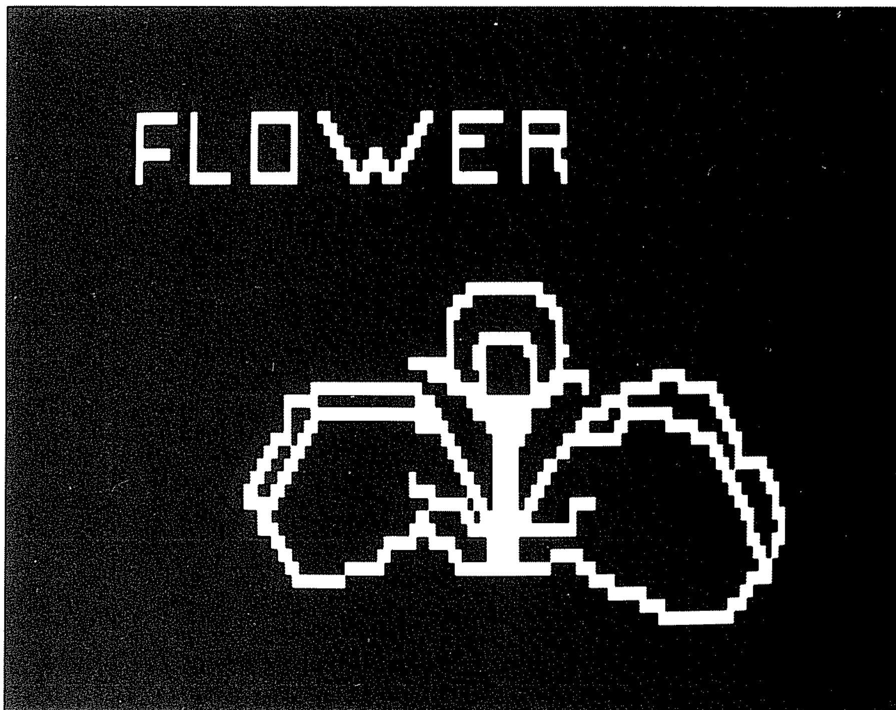


Photo 10-C-6

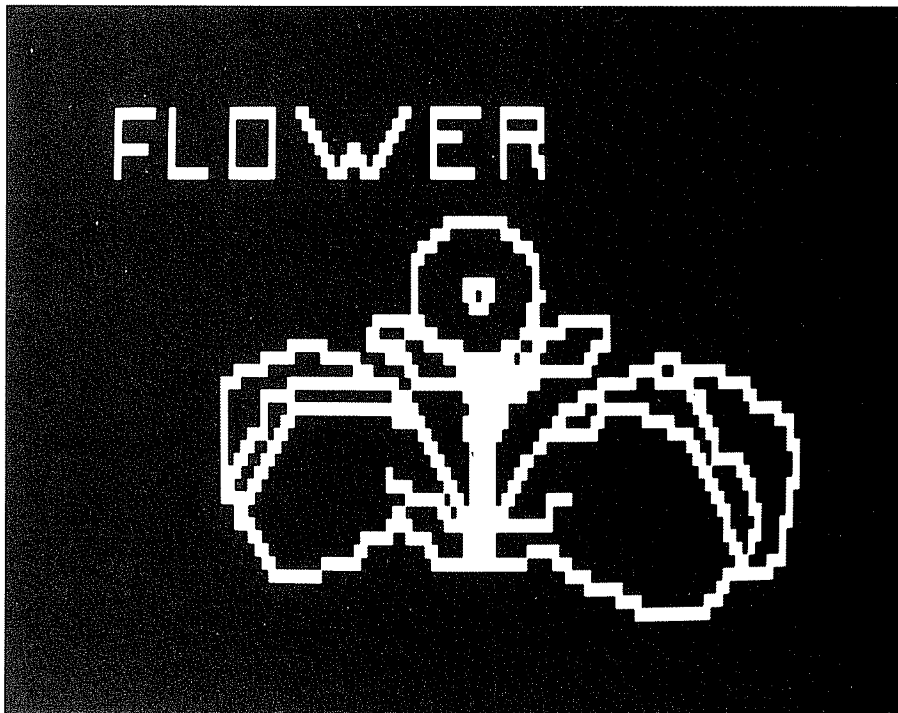


Photo 10-C-7

CONCLUSION

This program shows many techniques of graphics and file manipulation that are useful for graphics programming. Beyond that, it is a useful tool that can be used to create attractive screens for animated displays.

SUMMARY

Storing all the data for a screen on a disk file allows the user to access enough graphics for the screen quickly without storing it in the program itself.

If several of these screenfuls of data are loaded into memory at once, they can be moved to the video memory very quickly in sequence to create animation.

The following disk operating system (DOS) calls can be made to facilitate disk file manipulation in assembly language:

OPEN opens the file.
POSN positions the disk drive head at the proper file record.
READ transfers one record from the file into the buffer.
WRITE transfers one record from the buffer to the file.
CLOSE closes the file.

Meeting the Chapter Objectives

Here are the chapter objectives for your review. Can you meet all the objectives?

A. Write an explanation of why a programmer would want to use file handling to program graphics.
B. Write an explanation of the function, location, and conditions under which each of the following DOS calls are used: OPEN, POSN, READ, WRITE, and CLOSE.
C. Write a description of and the function of the Device Control Block (DCB).
D. Write an explanation of how a program can read data from the video memory into a data file.
E. Write an explanation of how a program can read data from a data file into the video memory.

More Programming Practice

1. Write a BASIC program which calls the machine language program in Listing 10-A and allows the user to "page" forward and backwards through an existing disk file.

2. Change the assembly language program in Listing 10-A so it reads only one record at a time from the disk file and displays it in the

top quarter of the video display.

3. Change the assembly language program in Listing 10-A so the entire memory of the computer is saved to a disk file.

4. Write an assembly language program which works independently (without being called by a BASIC program) to open a file called "DATA," read the first record into the buffer, move the data from the buffer to the screen, and close the file.

5. Write an assembly language program which accomplishes the same things as the program mentioned in item 4, and also has its DCB and buffer in the video memory.

Chapter Checkup

1. What conditions must exist before the OPEN DOS call can be made? The CLOSE call?

2. What conditions must exist before the POSN DOS call can be made?

3. What conditions must exist before the READ DOS call can be made? The WRITE call?

4. What does the GETARG ROM call do?

5. In the program in Listing 10-A, which program line moves the data which has been read from the disk into the buffer, to the screen?

6. In the program in Listing 10-A, which program lines check to see if whether enough of the data from the disk file has been displayed on the screen?

7. In the second half of the program in Listing 10-A, which lines load the correct number in BC before the POSN call is made?

8. In the program in Listing 10-C, line 1280 contains the following instruction: **PRINT@ INT((Y)/3)*64+INT(X/2),"";**. What is the function of this instruction?

9. Part of this program moves data from different areas of memory to the video memory to create animation. Which lines of the program in Listing 10-C contain the loop which specifies the proper memory location variables for these block moves?

10. What do lines 1800 to 1830 in the program in Listing 10-C do?

Appendices

Appendix 1: Video Display Worksheet/227

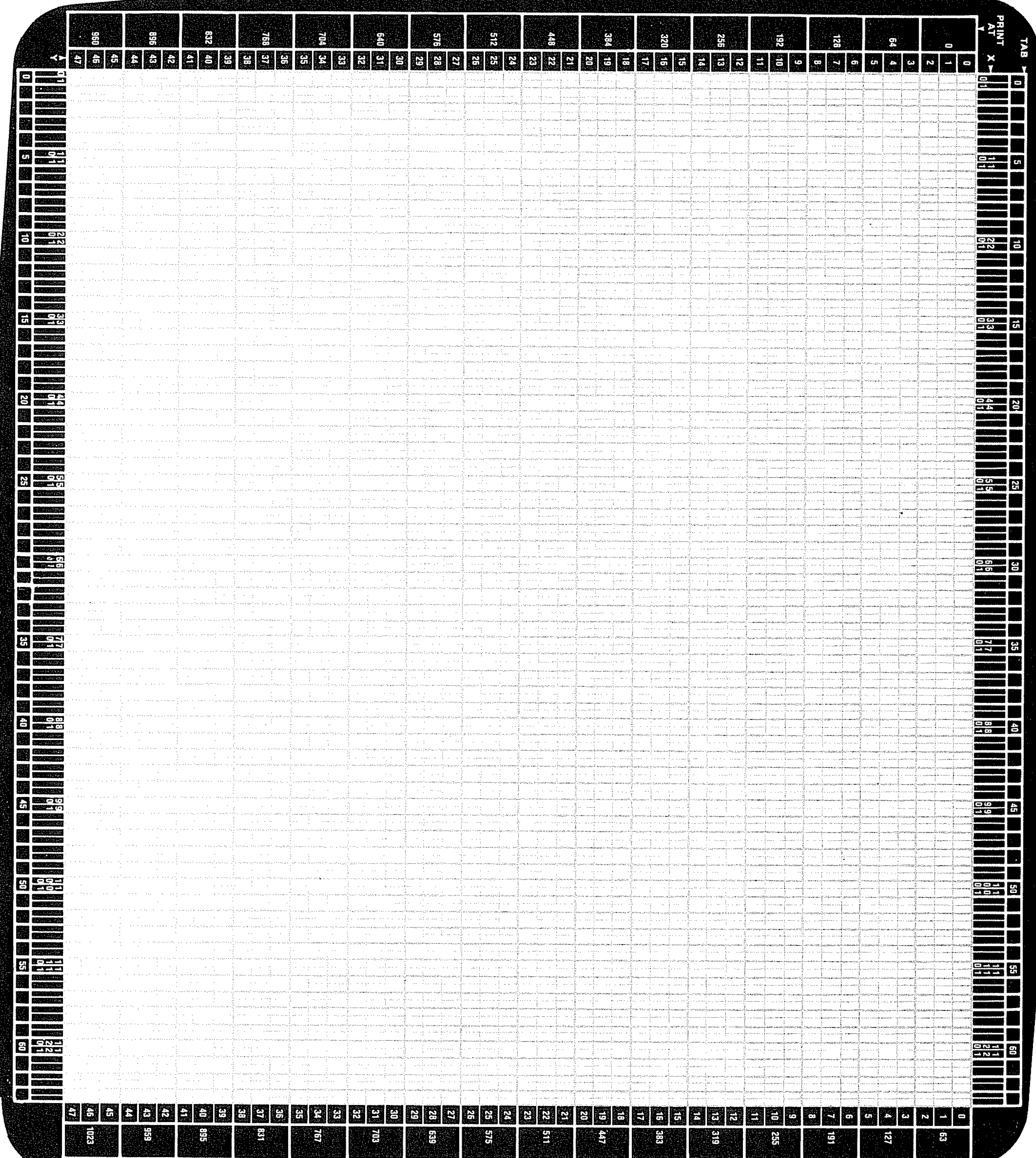
Appendix 2: Text Characters and Codes/228

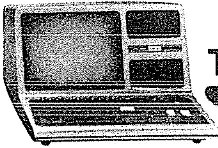
Appendix 3: Graphic Characters: 232

(Reproduced with permission from TRS-80 Model III Operation and Basic Reference Manual, Catalog No. 26-2111. © 1980, Tandy Corporation)

TRS-80 Video Display Worksheet

TITLE ______________________ PROGRAMMER ______________ PAGE ___ OF ___

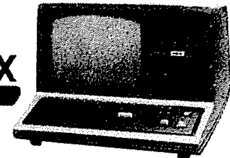




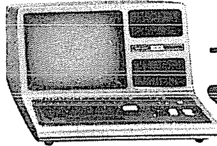
In the following table, we summarize the keyboard and video display control characters.

Code Dec.	Code Hex.	Keyboard †	Video Display PRINT CHR$ *(code)*
1	00		No effect
1	01	BREAK SHIFT ↓ A	No effect
2	02	SHIFT ↓ B	No effect
3	03	SHIFT ↓ C	No effect
4	04	SHIFT ↓ D	No effect
5	05	SHIFT ↓ E	No effect
6	06	SHIFT ↓ F	No effect
7	07	SHIFT ↓ G	No effect
8	08	← SHIFT ↓ H	Backspace and erase
9	09	→ SHIFT ↓ I	Tab (0, 8, 16, 24, . . .)
10	0A	↓ SHIFT ↓ J	Move cursor to start of next line and erase line
11	0B	SHIFT ↓ K	No effect
12	0C	SHIFT ↓ L	No effect
13	0D	ENTER SHIFT ↓ M	Move cursor to start of next line and erase line
14	0E	SHIFT ↓ N	Cursor on
15	0F	SHIFT ↓ O	Cursor off
16	10	SHIFT ↓ P	No effect
17	11	SHIFT ↓ Q	No effect
18	12	SHIFT ↓ R	No effect
19	13	SHIFT ↓ S	No effect
20	14	SHIFT ↓ T	No effect
21	15	SHIFT ↓ U	Swap space compression/ special characters
22	16	SHIFT ↓ V	Swap special/alternate characters
23	17	SHIFT ↓ W	Double-size characters
24	18	SHIFT ← SHIFT ↓ X	Backspace without erasing
25	19	SHIFT ↓ Y	Advance cursor
26	1A	SHIFT ↓ Z	Move cursor down
27	1B	SHIFT ↑	Move cursor up
28	1C	SHIFT ↓ ,	Move cursor to upper left corner
29	1D	SHIFT ↓ 9	Erase line and start over
30	1E	SHIFT ↓ .	Erase to end of line
31	1F	CLEAR SHIFT ↓ /	Erase to end of display

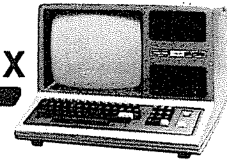
† Some of these keyboard characters can only be input using the INKEY$ function.



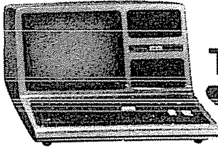
Code		Key-board	Video Display	
Dec.	Hex.		PRINT CHR$ *(code)*	POKE *vidram, code*
32	20	SPACEBAR	b̸	b̸
33	21	!	!	!
34	22	"	"	"
35	23	#	#	#
36	24	$	$	$
37	25	%	%	%
38	26	&	&	&
39	27	'	'	'
40	28	(	(	(
41	29	)	)	)
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	–	–	–
46	2E	.	.	.
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	:
59	3B	;	;	;
60	3C	<	<	<
61	3D	=	=	=
62	3E	>	>	>
63	3F	?	?	?
64	40	@	@	@
65	41	A	A	A
66	42	B	B	B
67	43	C	C	C
68	44	D	D	D



Code		Key-board	Video Display	
Dec.	Hex.		PRINT CHR$ *(code)*	POKE *vidram, code*
69	45	E	E	E
70	46	F	F	F
71	47	G	G	G
72	48	H	H	H
73	49	I	I	I
74	4A	J	J	J
75	4B	K	K	K
76	4C	L	L	L
77	4D	M	M	M
78	4E	N	N	N
79	4F	O	O	O
80	50	P	P	P
81	51	Q	Q	Q
82	52	R	R	R
83	53	S	S	S
84	54	T	T	T
85	55	U	U	U
86	56	V	V	V
87	57	W	W	W
88	58	X	X	X
89	59	Y	Y	Y
90	5A	Z	Z	Z
91	5B	⇧	[	[
92	5C		\	\
93	5D		]	]
94	5E		^	^
95	5F		_	_
96	60	SHIFT @	`	`
97	61	A	a	a
98	62	B	b	b
99	63	C	c	c
100	64	D	d	d
101	65	E	e	e
102	66	F	f	f
103	67	G	g	g
104	68	H	h	h
105	69	I	i	i



Code		Key-board	Video Display	
Dec.	Hex.		PRINT CHR$ *(code)*	POKE *vidram, code*
106	6A	J	j	j
107	6B	K	k	k
108	6C	L	l	l
109	6D	M	m	m
110	6E	N	n	n
111	6F	O	o	o
112	70	P	p	p
113	71	Q	q	q
114	72	R	r	r
115	73	S	s	s
116	74	T	t	t
117	75	U	u	u
118	76	V	v	v
119	77	W	w	w
120	78	X	x	x
121	79	Y	y	y
122	7A	Z	z	z
123	7B		{	{
124	7C		¦	¦
125	7D		}	}
126	7E		~	~
127	7F		±	±
128	80	Codes 128-191 output graphics characters. See the graphic display table in this Appendix.		
192 : :	C0	Codes 192-255 output either space compression codes or special characters when used with PRINT CHR$ *(code)*.		
255	FF	They always output special characters when used with POKE *vidram, code*. See the special character table in this Appendix.		



Graphics Characters (Codes 128-191)

135	143	151	159	167	175	183	191
134	142	150	158	166	174	182	190
133	141	149	157	165	173	181	189
132	140	148	156	164	172	180	188
131	139	147	155	163	171	179	187
130	138	146	154	162	170	178	186
129	137	145	153	161	169	177	185
128	136	144	152	160	168	176	184

ASCII format for SAVEing BASIC programs, 135

assembly language

defined, 158-159
drawing lines using, 194-200
file handling using, 202-219
LDIR instruction, 168-173
reversing graphics characters using, 193-194
registers, 163
running from BASIC programs, 165-168
scrolling the display sideways using, 178-185
splitting the display using, 187-191

banners, printing on printer, 98-103

bar graph programs

using printer, 94-98
using video display, 94-98

BREAK key, 4

circles, SETting using BASIC, 12-16

control codes for printing

using BASIC, 34-37
using DEBUG, 140-145

DEBUG under TRSDOS

combining BASIC program lines, 148-151
control characters, 140-145
modifying PRINT instructions, 146-148
protecting BASIC program code, 151-154
string packing, 134-139

ellipse, SETting, 14-15

educational programs

addition with large characters, 83-89
graph-reading program, 78-83

error trapping in BASIC, 62-63

file handling for graphics in assembly language, 202-219

graphic characters on printer, 107-108

graphing program, 70-74

hexadecimal numbers, 161-163

keyboard input routine using BASIC, 51-61

LDIR instruction in assembly language, 168-173

lines, 4

assembly language, 194-200
POKEing in BASIC, 49-50
RESETting in BASIC, 7
SETting in BASIC, 4, 8-11

LPRINT instruction in BASIC, 91

machine language defined, 158-159 (see also assembly language)

POINT instruction in BASIC, 1, 16-19

POKE instruction in BASIC, 42, 47

POKEing lines in BASIC, 49-50

PRINT instruction in BASIC, 23

PRINTing text characters, 23-26

PRINTing graphic characters, 26-31

program structure in BASIC, 69-70

registers in assembly language programming, 163

RESET instruction in BASIC, 1, 7-8

RESETting lines, 7

reversing graphics characters in assembly language, 193-194

RND function in BASIC, 7, 8, 11

screen printing ("screen dump") in BASIC

graphics only, 104-107
text only, 103-104

scrolling the display sideways in assembly language, 178-185

SET instruction in BASIC, 1-3

SETting lines in BASIC, 4, 8-11

SETting circles in BASIC, 12-16

SIN instruction in BASIC (sine), 92

splitting the display in assembly language, 187-191

string-packing programs

using BASIC, 122-128, 128-130
using DEBUG, 134

string space in BASIC

discussed, 173
displayed, 173-174

string variables in BASIC, PRINTing, 31-34

STRING$ function in BASIC, 37-39, 42-43

trigonometric functions in BASIC, 12-13

VARPTR instruction in BASIC ("variable pointer"), 113

video display worksheet, 2 (see also Appendix 1)

video memory, 47-51

reading graphics from, 61-63
storing number in, 63-66

X,Y coordinates, 2

In business, educational, and personal software, the addition of graphics can make a significant difference in the attractiveness and user-friendliness of your programs. But programming graphics can be very difficult. Determining which pixel to set and which character to print to create a graphic may take hours.

Now Dennis F. Tanner solves these and other graphics programming problems for you — whether you are an advanced programmer or novice. Covering models I, III, and 4, *The TRS-80 Graphics Book* shows you how to make your programs come alive with graphics. And, you can do it without spending too much time or foregoing the nicer graphic displays.

Many carefully explained sample programs give you a solid background in graphics programming. You will find find thorough coverage of several different programming techniques and the advantages and disadvantages of each.

Tanner details how to create graphics with SET/RESET/POINT programming. He tells you how to use 'PRINT' as a graphics instruction with regular and graphics characters. The 'POKE' instruction and its graphic uses with the video memory of the computer are clearly explained. You will also find out everything you need to know about assembly language programming and its application to graphics programming.

This timely guide puts an arsenal of *new* techniques at your fingertips. You will discover easy-to-follow instructions on how to handle:

- compressed graphics using VARPTR INSTRUCTION and DEBUG under TRSDOS
- graphics using block moves
- screen-to-printer graphics
- screen manipulation and movement
- screen designs
- screen reverses
- line drawings
- simple animation

You will gain useful information on file handling in assembly language and its application to graphics programming. Included is a program with a screen editor that allows you to save files for later retrieval.

The TRS-80 Graphics Book provides you with practice programming challenges that help you put vital data right to work. It also gives you a means of checking your progress, and numerous utility programs that make graphic programming easier. Let it be your key to better, easier computer graphics.

About the Author

Dennis F. Tanner is Director of Educational Product Development, Tandy, Radio Shack. He lives and works in Fort Worth, Texas.

VAN NOSTRAND REINHOLD COMPANY
135 West 50th Street, New York, N.Y. 10020

ISBN 0-442-28299-0