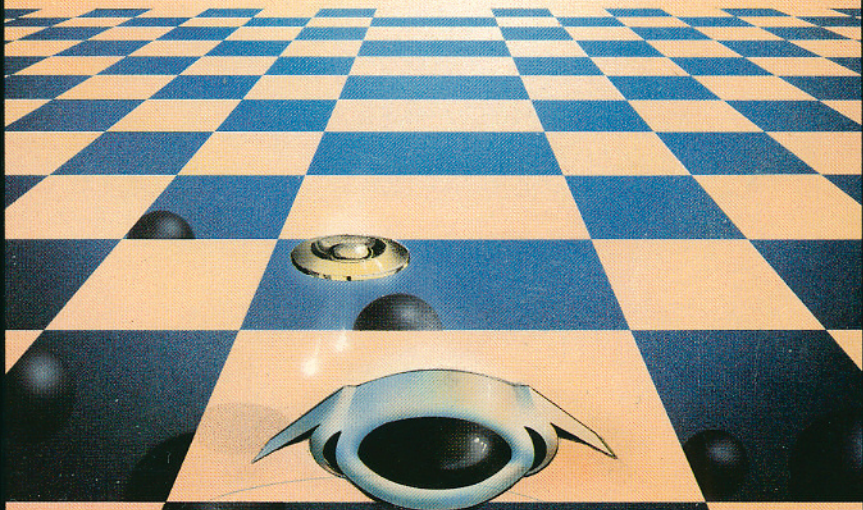


# SMART PROGRAMMING GUIDE™ FOR SPRITES



```
100 CALL CLEAR :: CALL SCREE  
N(12) :: FOR N=1 TO 4 :: CALL  
SPRITE(#N,64+N,2,100,100+8*  
N) :: NEXT N  
110 FOR N=5 TO 12 :: CALL SP  
RITE(#N,64+N,2,1,140,6,0) ::  
FOR T=1 TO 99 :: NEXT T :: N  
EXT N  
120 CALL PEEK(-31877,N) :: IF  
N AND 64 THEN DISPLAY AT(24  
,1)BEEP:USING "SPRITE ## IS  
FIFTH ON LINE":(N AND 31)+1  
ELSE CALL CLEAR  
130 GOTO 120
```

# SMART PROGRAMMING GUIDE™ FOR SPRITES

## *On the Texas Instruments 99/4 or 99/4A Computer*

By

Craig G. Miller

Copyright © 1983 by Millers Graphics

Minimum equipment requirements are a console, the extended basic command module and a monitor or TV set.

Millers Graphics makes no warranty, either expressed or implied, including, but not limited to, any implied warranties or merchantability and fitness for a particular purpose, regarding the materials in this book or any programs derived therefrom and makes such materials available solely on an "as is" basis. In no event shall Millers Graphics be liable to anyone for special, collateral, incidental or consequential damages in connection with the purchase or use of this book.

TI 99/4, TI 99/4A and Solid State Command Module are registered trademarks of Texas Instruments, Inc.

Copyright © 1983 by Millers Graphics  
All rights reserved.  
Printed in the United States of America.  
No part of this publication may be reproduced, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher.



---

In this Smart Programming Guide for Sprites we have set up a number of program examples to help you understand the nature of sprites. We strongly recommend that you start at the beginning and work your way through the book since many of the items that have been covered in the beginning are not as fully explained later on in the book.

In the documentation for each program we have used a double colon to indicate that we are referring to the next statement in a multiple statement line.

---

## TABLE OF CONTENTS

Smart Programming Tips.....	1
Conversion Formulas.....	3
Call Char Tips.....	5
Call Joyst.....	7
Joyst 1.....	7
Joyst 2.....	8
Joyst 3.....	10
Call Key.....	13
Key 1.....	13
Key 2.....	14
Key 3.....	16
Call Peek.....	19
0-99 Random Numbers.....	19
VDP Interrupt Timer.....	20
Highest Numbered Sprite.....	21
VDP Status Register.....	22
0-255 Double Random Numbers.....	25
Sprite Patterns.....	27
Pattern 1.....	28
Pattern 2.....	29
Pattern 3.....	30
Sprite Chase.....	31
Chase 1.....	31
Chase 2.....	33
Shooting Sprites.....	35
Pick up Objects.....	39
Pickup 1.....	40
Pickup 2.....	43
Eat Dots and Lay Down Trail.....	49
Dots.....	50
Maze Puzzle.....	57
General Bar Grapher.....	63

---

$M_G$  

---

With proper program structuring, the TI 99 home computer with the extended basic command module can be very powerful. Listed below are some tips that will help you to save bytes and enhance the speed of your programs.

1. Keep your variable and string variable names as short as possible. Preferably 1 or 2 characters.
2. Keep the number of different variables and string variables to a minimum by reusing them whenever possible.
3. Regular variables such as A or A\$ run faster than subscripted variables such as A(2) or A\$(2) which run faster than arrayed variables such as A(2,3) or A\$(2,3).
4. Always try to use full multiple statement lines.
5. Always use the lowest possible numbers for your sprites that are or will be in motion.
6. Try to define as many characters as possible in each CALL CHAR statement.
7. Keep your GOTO and GOSUB statements to a minimum.
8. The fewer the bytes in a statement, the faster it will run, is a good general rule to remember when programming.



---

## NOTES

---

---

# CONVERSIONS

---

---

The following conversion formulas can be included in your programs whenever you need to convert a sprite position into a graphic or text position or visa versa.

## CONVERSION FORMULAS

FROM	TO	
Graphic Row	Dot Row	$GR*8-7=DR$
Graphic Column	Dot Column	$GC*8-7=DC$
Text Row	Dot Row	$TR*8-7=DR$
Text Column	Dot Column	$TC*8+9=DC$
With Rounding Off		
Dot Row	Graphic Row	$(DR+7)/8=GR,$ <i>See note 1</i>
Dot Column	Graphic Column	$(DC+7)/8=GC,$ <i>See note 2</i>
Dot Row	Text Row	$(DR+7)/8=TR,$ <i>See note 1</i>
Dot Column	Text Column	$(DC-9)/8=TC,$ <i>See note 3</i>
Without Rounding		
Dot Row	Graphic Row	$INT((DR+7)/8)=GR,$ <i>See note 4</i>
Dot Column	Graphic Column	$INT((DC+7)/8)=GC$
Dot Row	Text Row	$INT((DR+7)/8)=TR,$ <i>See note 4</i>
Dot Column	Text Column	$INT((DC-9)/8)=TC,$ <i>See note 5</i>
GR = Graphic Row		
GC = Graphic Column		
DR = Dot Row		
DC = Dot Column		
TR = Text Row		
TC = Text Column		

---

## NOTES

- Note 1* The effective dot rows are 1 thru 188. Dot values of 189 thru 256 return 24.5 thru 32.875 and will be rounded up to 25 thru 33 which are not valid graphic or text rows and will cause an error message to be generated for graphic rows. See *note 3*.
- Note 2* The effective dot columns are 1 thru 252. Dot columns of 253 thru 256 return 32.5 thru 32.875 which will be rounded up to 33 which is not a valid graphic column and will cause an error message to be generated.
- If you change this formula to  $(\text{DOT COLUMN} + 3) / 8 = \text{GRAPHIC COLUMN}$ , all of the dot columns will be within the effective range. It will then return graphic columns from .5 thru 32.375 which will be rounded to 1 thru 32. However, this formula is not as accurate since it is displaced by 4 dot columns to the left of your sprites actual position.
- Note 3* The effective dot columns are 13 thru 236. Dot columns from 1 thru 12 return -1 thru .375 and will not round up to 1. Dot columns from 237 thru 256 return 28.5 thru 30.88 which when rounded off equal text columns 29 thru 31 and are not valid. Even though the low and high values are out of the text range when you use them as display at or tab values they will not cause your program to halt with an error message, however, your displays will not be formatted properly on the screen.
- Note 4* The effective dot rows are 1 thru 192. Dot rows 193 thru 256 return 25 thru 32 which are not valid rows and will cause an error message to be generated.
- Note 5* The effective dot columns are 17 thru 240. Dot columns 1 thru 16 return 0 and dot columns 241 thru 256 return 29 thru 30 which are not valid text columns. See *note 3*.

Here are a few tips on using CALL CHAR.

1. You may define more than 4 characters or characters that are not in numerical sequence in a single CALL CHAR statement.  
EXAMPLE: 100 CALL CHAR(65,A\$,73,B\$,93  
"FF00F",70,"10AFCE",96,A\$,99,B\$)
2. Using string-variables such as A\$ or direct strings such as "FF00FF00FF" in the pattern identifier section of your CALL CHAR statement will not make a difference in its speed of operation.
3. When defining a character for use in CALL MAGNIFY 3 or 4 sprites (256 dots or 4 characters) your character-code (character number) must be divisible by 4 without a remainder, such as 96,100,104.
4. Sprite characters do not have to be defined in separate character sets. You may define character 96 as a red wall on a yellow background and character 97 as a ghost and set its color as white in your CALL SPRITE statement.
5. Redefining character numbers 33-43, 58-64 and 91-95 as sprite characters will allow you to use all of the characters in sets 9 thru 14 for screen graphics and you will still have all of the alphabet and numbers for text display.
6. Try to place your CALL CHAR statements in the beginning of your program where you need slight delays such as when the title is on the screen. This is also a good time to assign values to your variables and define your strings. By doing this you can cut down on the apparent initialization time of your program.

---

## NOTES

The following three programs were written to demonstrate different ways of moving your sprites around the screen. The first program sets your sprite in motion only when the joystick is moved. The second program demonstrates additive motion. The longer you hold the stick in one direction, the faster your sprite will move. To stop the sprite you will need to cancel out the motion by holding the stick in the opposite direction. The third program moves your sprite to a new graphic row and/or column each time the stick is moved.

All eight positions on the number one joystick are active for these examples. (See note at the end of third joystick program documentation.)

## JOYST 1

```
100 CALL CLEAR :: CALL SCREE
N(13):: CALL MAGNIFY(2):: CA
LL SPRITE(#1,42,16,100,100)
```

```
110 CALL JOYST(1,X,Y):: CALL
MOTION(#1,-Y*4,X*4):: DISPL
AY AT(23,4):"Y=";Y," X=";X:
"-Y*4=";-Y*4,"X*4=";X*4 :: G
OTO 110
```

- 100 Clears the screen:: Changes screen color to dark green:: Sets sprites to magnification of 2, a single character made up of up to 64 dots which occupies 4 character positions on the screen:: Sets sprite number 1 as character number 42, the asterisk, its color will be white and it will be placed at dot row 100 and dot column 100, the velocities were left out so the sprite will be stationary.

---

110 Now that our sprite is set up on the screen, our program will only loop on this line. The first thing we will do is to look for a joystick input. If the number one joystick is moved then X or Y or both will contain -4,0 or 4 depending on how the stick is moved. The Y variable will contain the row velocity commands and the X variable will contain the column velocity commands of the CALL JOYST statement. The statement will place our sprite in motion according to the position of the joystick. If the joystick has not been moved then X and Y will equal 0 and our sprite will stop. If we push the stick up then X will equal 0 and Y will equal 4. Our sprite needs negative row velocities to move up and positive velocities to move down so we will have to change the value of Y by placing a minus sign before it. Now when Y equals 4, -Y will equal -4 and when Y equals -4, -Y will equal 4. The column velocity commands do not have to be changed around since X will contain -4 when the stick is moved left and 4 when it is moved right and this will move our sprite in the proper direction. In order to speed our sprite up, we will multiply our X and -Y values by 4 to obtain velocities of -16,0 or 16. If you would like to use different velocities you can change the 4 to any positive number that is less than or equal to 31.75. A number that is higher than 31.75 will return a positive row or column velocity that is greater than 127 and will cause an error message to be generated. In this section of line 110 we will display the X and Y values from the CALL JOYST statement and the sprite velocities as you move the stick to different positions. Now we will jump back to the beginning of line 110 and do it all over again.

## JOYST 2

```
100 CALL CLEAR :: CALL SCREE
N(13):: CALL MAGNIFY(2):: CA
LL SPRITE(#1,42,16,100,100)
```

```
110 CALL JOYST(1,C,R):: X=(X
+C)*-(ABS(X)<124):: Y=(Y-R)*
-(ABS(Y)<124):: CALL MOTION(
#1,Y,X):: DISPLAY AT(24,1):"
Y=";Y,"X=";X :: GOTO 110
```

- 
- 100 Clears the screen:: Changes screen color to dark green:: Sets sprites to a magnification of 2, a single character made up of up to 64 dots which occupies 4 character positions on the screen:: Sets sprite number 1 as character number 42, the asterisk, its color will be white and it will be placed at dot row 100 and dot column 100, the velocities were left out so the sprite will be stationary.
- 110 Now we will look for a joystick input and place - 4,0 or 4 into C and/or R:: This group of statements is the same as **IF THE ABSOLUTE VALUE OF X IS LESS THAN 124 THEN X = X+C ELSE X=0**. Lets say the value of X is 120 and that C equals 4. Since the computer completes all of its tests and functions before it changes the value of X, X will be equal to 120 in both of the bracketed statements. The (X+C) statement will then equal 124, (120+4). The second statement is a test. If the absolute value of X is less than 124, which it is right now, then everything in the brackets is true and it will equate out to -1. If it was false then it would become 0, so this test statement will only return values of -1 for true or 0 for false. Since it is true then X equals (X+C)\*(-1) or  $X=(124)*(-1)$  or  $X=124$ . Now when the program loops thru here again X will not be less than 124 so  $X=(X+C)*(0)$  equals 0. This is to prevent X from having an absolute value greater than 127 which would cause an error message to be generated:: This group of statements is the same as the previous group, with the exception of Y-R. By subtracting R from Y we are able to change the -4 when the stick is pushed down into +4 and the +4 when pushed up into -4:: So far we have looked for a joystick input, tested the values of X and Y to make sure they are within the proper limits and changed their values according to the test or the joystick movement. Now we will change the speed of our sprite to correlate with the values of Y and X:: This statement will allow us to watch the values of X and Y change on the screen as we move the joystick around:: Now we will jump back to the beginning of line 110 and do it all over again.



---

## JOYST 3

```
100 CALL CLEAR :: CALL COLOR
(2,7,7):: CALL SCREEN(11)::
CALL HCHAR(24,1,40,64):: CAL
L VCHAR(1,31,40,96):: CALL S
PRITE(#1,42,2,17,17):: R,C=3

110 CALL JOYST(1,X,Y):: X=SG
N(X):: Y=-SGN(Y)

120 CALL GCHAR(R+Y,X+C,CH)::
IF CH=40 THEN CALL SOUND(-6
0,110,9):: GOTO 110 ELSE C=C
+X :: R=R+Y :: CALL LOCATE(#
1,R*8-7,C*8-7):: GOTO 110
```

- 100 Clears the screen:: Changes the color of set 2 to a dark red foreground on a dark red background. All the characters, in set 2 numbers 40 thru 47 will now be dark red blocks:: Changes the screen color to dark yellow. Now we will place some walls on the screen. The CALL HCHAR statement places our blocks character number 40, starting at row 24, column 1, 64 times. Since there are only 32 columns on the screen, repetition numbers 33 thru 64 will wrap around to the top of the screen. This forms the top and bottom walls:: The CALL VCHAR statement places out blocks starting at row 1, column 31, 96 times. Since there are only 24 rows on the screen, repetition numbers 1 thru 48 will place two full columns of blocks on the right hand edge of the screen and repetition numbers 49 thru 96 will wrap around to the left hand edge of the screen and form two more full columns:: This will set up sprite number 1 as character 42, the asterisk, its color will be black and it will be at dot row 17 and dot column 17. Dot row 17 and dot column 17 are the same as graphic row 3 and graphic column 3,  $3*8-7=17$ :: Now we will set our starting graphic row and column positions equal to 3.

- 
- 110 Now that everything is set up our program will only loop thru lines 110 and 120. First we will look for a joystick input and place -4,0 or 4 into X and/or Y:: With this function we can change any positive value of X into 1, any negative value of X into -1 and if X equals 0 then  $\text{SGN}(X)$  will equal 0. So we have now changed the -4,0 or 4 the **CALL JOYST** statement returns into -1,0 or 1:: This is the same as the last statement except we have placed a minus sign before the function to obtain a -1 value when the stick is pushed up and a +1 value when pushed down.
- 120 This statement will make the computer get the screen character at the location that we want to move our sprite to. Let's say the program has just started running and we have moved the joystick to the left. Our sprite is currently at row 3 and column 3 so R and C equal 3. Since we have pushed the stick left Y will equal 0 and X will equal -1 and we will get the screen character at row 3, (3+0) and at column 2, (3+-1) and place it in the variable CH. Since our wall is in column 2, CH will equal 40. If we would have pushed the stick right CH would have equaled 32, the space character:: So, if CH equals 40, then we will generate a low frequency sound:: and jump back up to line 110 without moving our sprite to that screen location. If CH does not equal 40 then we will add the value of X to C:: And add the value of Y to R:: Let's say you have moved the stick to the right, then C will equal 4 and R will equal 3, so, X+C equals 4 and Y+R equals 3. Now we will move our sprite to its new location but we have to change the graphic row and column values into dot row and column values,  $3*8-7=17$ ,  $4*8-7=25$  so our sprite will be relocated to dot row 17, the same row it started on and dot column 25, 1 graphic column to the right:: Jump back to line 110 and do it all over again.

**NOTE:**

If you want to eliminate the diagonal moves the **CALL JOYST** statement returns in the previous examples change 110 to read as follows:  
110 **CALL JOYST(1,X,Y):: IF X AND Y THEN 110 ELSE ...**  
(rest of 110). The **IF X AND Y** statement is the same as **IF X<>0 AND Y<>0** and will only be true when the stick is moved diagonally. You can also change it to **IF X\*Y THEN 110 ELSE ...** and obtain the same results.

---

## NOTES

The following three programs were written to demonstrate different ways of moving your sprites around the screen with the keyboard. The first program sets your sprite in motion only when a key is pressed. The second program demonstrates additive motion. The longer you hold down a key, the faster your sprite will move. To stop the sprite you will need to cancel out the motion by holding down a key in the opposite direction. The third program moves your sprite to a new graphic row and/or column each time you press a key. The arrow keys ESD and X and the diagonal keys WRZ and C are active in all three programs. (*See note at the end of the third program's documentation.*) If the program does not react properly when you press the down arrow key X, then change  $K=0$  to  $K+1=1$  in the test sections after CALL KEY in line 110 of each program.

### KEY 1

```
100 CALL CLEAR :: CALL SCREE
N(13):: CALL MAGNIFY(2):: CA
LL SPRITE(#1,42,16,100,100)
```

```
110 CALL KEY(1,K,S):: Y=(K>3
AND K<7)-(K=0 OR K=15 OR K=
14):: X=(K=2 OR K=4 OR K=15)
-(K=3 OR K=6 OR K=14)
```

```
120 CALL MOTION(#1,Y*16,X*16
):: DISPLAY AT(23,9):"KEY=";
K:"Y*16=";Y*16,"X*16=";X*16
:: GOTO 110
```

- 100 Clears the screen:: Changes screen color to dark green:: Sets sprites to a magnification of 2, a single character made up of up to 64 dots which occupies 4 character positions on the screen:: Sets sprite number 1 as character number 42, the asterisk, its color will be white and it will be placed at dot row 100 and dot column 100, the velocities were left out so the sprite will be stationary.

- 110 First we will look for a key press on the left hand side of the keyboard and place its value in the variable K:: This statement is the same as IF K=4 OR K=5 OR K=6 THEN Y=-1 ELSE IF K=0 OR K=15 OR K=14 THEN Y=1 ELSE Y=0. Let's say you have pressed the C key and K now equals 14. Since K does not equal 4, 5 or 6 the first bracketed group equals 0 or false. Since K equals 14 the second bracketed group equals -1 or true and the whole thing appears like this; Y=(0)-(-1). Minus a minus 1 equals 1 so Y=1:: This statement is the same as IF K=2 OR K=4 OR K=15 THEN X=-1 ELSE IF K=3 OR K=6 OR K=14 THEN X=1 ELSE X=0. If we continue with K equaling 14 then this statement appears like this X=(0)-(-1) or X=1. So at this point in the program if we have pressed the C key both X and Y would equal 1. Which is the proper value for moving our sprite diagonally down and to the right. These two statement groups will make X and/or Y equal to -1,0 or 1 depending on which, if any, key is pressed.
- 120 Now that we have the proper X and Y values from our key press we can set our sprite in motion. Since a velocity of 1 is slow we will multiply our X and Y values by 16:: Now we will display the value of K and the sprite row and column velocities on the screen:: Jump back to 110 and do it all over again.

## KEY 2

```
100 CALL CLEAR :: CALL SCREE
N(13):: CALL MAGNIFY(2):: CA
LL SPRITE(#1,42,16,100,100)
```

```
110 CALL KEY(1,K,S):: IF S T
HEN Y=Y+4*((K>3 AND K<7)-(K=
0 OR K=15 OR K=14)):: X=X+4*
((K=2 OR K=4 OR K=15)-(K=3 O
R K=6 OR K=14))
```

```
120 Y=Y*-(ABS(Y)<128):: X=X*
-(ABS(X)<128):: CALL MOTION(
#1,Y,X):: DISPLAY AT(24,1):"
Y=";Y,"X=";X :: GOTO 110
```

- 
- 100 Clears the screen:: Changes screen color to dark green:: Sets sprites to a magnification of 2, a single character made up of up to 64 dots which occupies 4 character positions on the screen:: Sets sprite number 1 as character number 42, the asterisk, its color will be white and it will be placed at dot row 100 and dot column 100, the velocities were left out so the sprite will be stationary.
- 110 First we will scan the left side of the keyboard and look for a key press. If we press a key then K will contain its value and S will equal 1, -1 if the key was held down for more than one loop thru lines 110 and 120:: The IF S statement is the same as IF S<>0 and is only true when a key is pressed. This test was placed in the program to keep your sprite moving smoothly. If you hold down the 1 key while the program is running and your sprite is in motion you will bypass the test and the motion will not be as smooth. If this test is true then your program will execute the THEN statements. These statements are very similar to the statements in line 110 of the previous example except that now we will multiply the results of the tests by 4 and add this to the previous value of Y. So the first group will convert itself into  $Y=Y+4*((0)-(0))$  which equals  $Y=Y+0$ , if none of the active keys were pressed or into  $Y=Y+4*((-1)-(0))$  which equals  $Y=Y-4$ , or  $Y=Y+4*((0)-(-1))$  which equals  $Y=Y+4$ , if one of the active keys were pressed:: This test group is the same as the Y test group except now we will change the value of X according to the key pressed.
- 120 Now that we have converted our key press into -4, 0 or 4 and added this to X and or Y we will test to make sure our new X and Y values are within the proper limits. So if the absolute value of Y is less than 128 this test will equate out to  $Y=Y*(-1)$  which equals  $Y=Y*1$ . If Y is equal to 128 then our test equates out to  $Y=Y*(0)$  or  $Y=0$ :: This test group is the same as the last one except now we will make sure that X is within the proper limits:: Now we will change the motion of our sprite to correlate with the values of Y and X:: This statement will allow us to watch the values of Y and X change on the screen as we press keys:: Now we will jump back to line 110 and do it all over again.

---

## KEY 3

```
100 CALL CLEAR :: CALL COLOR  
(2,7,7):: CALL SCREEN(11)::  
CALL HCHAR(24,1,40,64):: CAL  
L VCHAR(1,31,40,96):: CALL S  
PRITE(#1,42,2,17,17):: R,C=3
```

```
110 CALL KEY(1,K,S):: Y=(K>3  
AND K<7)-(K=0 OR K=15 OR K=  
14):: X=(K=2 OR K=4 OR K=15)  
-(K=3 OR K=6 OR K=14)
```

```
120 CALL GCHAR(R+Y,X+C,CH)::  
IF CH=40 THEN CALL SOUND(-6  
0,110,9):: GOTO 110 ELSE C=C  
+X :: R=R+Y :: CALL LOCATE(#  
1,R*8-7,C*8-7):: GOTO 110
```

- 100 Clears the screen:: Changes the color of set 2 to a dark red foreground on a dark red background. All the characters, in set 2 numbers 40 thru 47 will now be dark red blocks:: Changes the screen color to dark yellow:: Now we will place some walls on the screen. The CALL HCHAR statement places our blocks character number 40, starting at row 24, column 1, 64 times. Since there are only 32 columns on the screen repetition numbers 33 thru 64 will wrap around to the top of the screen. This forms the top and bottom walls:: The CALL VCHAR statement places our blocks starting at row 1, column 31, 96 times. Since there are only 24 rows on the screen repetition numbers 1 thru 48 will place two full columns of blocks on the right hand edge of the screen and repetition numbers 49 thru 96 will wrap around to the left hand edge of the screen and form two more full columns:: This will set up sprite number 1 as character 42, the asterisk, its color will be black and it will be at dot row 17 and dot column 17. Dot row 17 and dot column 17 are the same as graphic row 3 and graphic column 3,  $3*8-7=17$ :: Now we will set our starting graphic row and column positions equal to 3.

- 
- 110 First we will scan the left hand side of the keyboard and look for a key press.:: Now we will test to see if we have pressed an active key. These two statement groups are exactly the same as the first CALL KEY example, so the final values we can obtain for X and/or Y will be -1,0 or 1. Let's look at this one more time.

No active key pressed equates to  $Y=(0)-(0)::X=(0)-(0)$ .

W (4) key pressed,  $Y=(-1)-(0)::X=(-1)-(0)$ .

E (5) key pressed,  $Y=(-1)-(0)::X=(0)-(0)$ .

R (6) key pressed,  $Y=(-1)-(0)::X=(0)-(-1)$ .

S (2) key pressed,  $Y=(0)-(0)::X=(-1)-(0)$ .

D (3) key pressed,  $Y=(0)-(0)::X=(0)-(-1)$ .

Z (15) key pressed,  $Y=(0)-(-1)::X=(-1)-(0)$ .

X (0) key pressed,  $Y=(0)-(-1)::X=(0)-(0)$ .

C (14) key pressed,  $Y=(0)-(-1)::X=(0)-(-1)$ .

- 120 Now that we have set up the values of Y and/or X according to the key pressed we will make the computer get the screen character at the location we want to move our sprite to. Since our wall is made up of character number 40, if we press a key that would move our sprite into a wall then CH will equal 40.:: Now we will test to see if CH equals 40 and if it does we will generate a low frequency sound.:: and jump back to line 110, without moving our sprite to look for another key press. ELSE we will add Y to R.:: and X to C.:: and move our sprite to its new location. Since R and C keep track of the graphic row and column position of our sprite we will need to convert these values into dot row and dot column values when we move our sprite.:: Now we will jump back up to line 110 and do it all over again.

#### NOTE:

If you want to eliminate the diagonal moves you can change line 110 as follows for the first and third examples;

110 CALL KEY(0,K,S)::  $Y=(K=5)-(K=0)::X=(K=2)-(K=3)$

AND TO: 110 CALL KEY(0,K,S)::  $Y=Y+4*((K=5)-(K=0))::$

$X=X+4*((K=2)-(K=3))$

For the second example.



---

## NOTES

```
100 RANDOMIZE :: CALL PEEK(-  
31880,A):: PRINT A;:: GOTO 1  
00
```

You must execute a **RANDOMIZE** statement prior to peeking into this address to obtain a random number. Generating random numbers through this address is much faster than the **RND** statement. The above program does not demonstrate the increased speed because it is slowed down by the **PRINT** statement. There is an example at the end of this section that demonstrates the speed difference between **RND** and **CALL PEEK** random number generators. The other advantage to this method is that it only generates integers between 0 and 99 so if you add 1 to it, it will generate integers between 1 and 100.

---

## VDP INTERRUPT TIMER

- 31879 VDP (video display processor) interrupt timer. This address increments itself every sixtieth of a second and generates sequential integers from 0 thru 255.

```
100 CALL PEEK(-31879,A):: IF  
    A<5 THEN PRINT : :A;:: GOTO  
    100 ELSE PRINT A;:: GOTO 10  
0
```

*Note: For version 100 extended basic  
change IF A<5 THEN ... to IF A< 7  
THEN ...*

This program was set up to skip a PRINT line each time this address is reset. The VDP interrupt timer is reset every 4.25 seconds, ( $255/60=4.25$ ) but with all the PRINT statements you will find that it resets approximately every 4.8 seconds.

---

## HIGHEST NUMBERED SPRITE

-31878 This address contains the highest numbered sprite that is in motion. In the old extended basic command modules (VERSION 100) this address is always loaded with 28. In the new extended basic command modules (VERSION 110) this address is updated everytime a sprite is put into motion or a sprite is deleted. Type in **CALL VERSION(A):: PRINT A** to find out which version of extended basic you have. This is one of the main factors in the speed difference between the old extended basic and the new, and faster extended basic. VERSION 100 always tries to update 28 sprites whether or not there are any on the screen and VERSION 110 will only update the number of sprites in this address. Here are a few tips on sprite use for maximum program speed with VERSION 110.

1. Always use the lowest possible numbered sprites for motion.
2. Even when you have placed zero's in the row and column velocity section of **CALL SPRITE** or the **CALL MOTION** the computer will still presume that sprite has to be updated.
3. A sprite that is placed on the screen and moved around with **CALL LOCATE** will not need motion updating and it will not effect the value in this address.
4. **CALL DELSPRITE(ALL)** will reset this address to 0. **CALL DELSPRITE(#X)** will adjust this address to the next highest sprite number that is in motion, if X was the highest one deleted.

---

## VDP STATUS REGISTER

- 31877 This address is a copy of the VDP status register. It contains information on sprite coincidence and if there are more than 4 sprites on a row it contains the number of the 5th (invisible) sprite. Like all registers this register is made up of 8 bits (1 byte), and to obtain useful information from this status register we need to know which bits are on (1) and which ones are off (0). The bits are numbered from 0 to 7 and correlate with binary positions that have set decimal values.

Bit #	0	1	2	3	4	5	6	7
Decimal Value	128	64	32	16	8	4	2	0

- Bit 0 is the 60 HZ VDP interrupt.  
Bit 1 is on anytime there are 5 or more sprites on the same row.  
Bit 2 is on anytime there is a sprite coincidence. (Same as CALL COINCIDENCE(ALL,C)).  
Bit 3-7 will contain the number of the 5th sprite on a row when bit 1 is on.

```
100 CALL CLEAR :: CALL SCREE
N(12):: FOR N=1 TO 4 :: CALL
  SPRITE(#N,64+N,2,100,100+8*
N):: NEXT N
```

```
110 FOR N=5 TO 12 :: CALL SP
RITE(#N,64+N,2,1,140,6,0)::
FOR T=1 TO 99 :: NEXT T :: N
EXT N
```

```
120 CALL PEEK(-31877,N):: IF
N AND 64 THEN DISPLAY AT(24
,1)BEEP:USING "SPRITE ## IS
FIFTH ON LINE":(N AND 31)+1
ELSE CALL CLEAR
```

```
130 GOTO 120
```

- 
- 100 Clears the screen:: Changes the screen color to light yellow:: Now, we will loop here 4 times:: and set up sprite numbers 1,2,3, and 4 as A,B,C, and D at dot row 100 and dot columns 108, 116, 124, and 132 respectively. Since we left out the velocity parameters these sprites will all be stationary near the middle of the screen on the same row:: Increments **N** four times and leaves the loop.
- 110 Now, we will loop here 8 times:: and set up sprite numbers 5,6,7,8,9,10,11, and 12 as E,F,G,H,I,J,K and L. We will generate these sprites at dot row 1 and dot column 140 with a row velocity of 6 and a column velocity of 0. So our sprites will travel straight down:: Now, we will loop here 99 times to kill some time so that the previously generated sprite can move down the screen a little before we generate the next sprite:: Increment **T** 99 times:: and then increment **N** and generate the next sprite, when **N** is greater than 12 we will leave this loop.
- 120 Now, our program will only loop thru lines 120 and 130, first we will **PEEK** into the copy of the VDP status register and put its value into **N**:: The **IF N AND 64** is the statement used to see if bit 1 is turned on. When you use logical operators (**AND**, **OR**, **XOR** and **NOT**) directly on numbers they are converted to binary form and each bit is tested according to the logical operator in use. Since 64 in binary form is 01000000, and we are using the **AND** operator we will be checking to see if **N** in binary form has a 1 in the corresponding binary position, which is bit 1. If it does then bit 1 is on (our statement is true) and there are 5 or more sprites on the line:: so, we will display our line of text with the **USING** statement and place the value we obtain from **(N AND 31)+1** into the **##** signs. 31 in binary form is 00011111 so, we will be comparing bits 3 thru 7 to see which ones are on. Whenever we use the **AND** operator on direct numbers, the computer changes our numbers into binary form and compares the bits in each binary position. If they match it, places a 1 in the corresponding binary position of another register and then converts this register back into decimal form for our use. (See Chapter 3 in your *TI Extended Basic* book for more information). Now, we will jump down to line 130:: If bit 1 is not turned on we will execute the **ELSE** and clear the text off the screen.
- 130 Jump back up to line 120 and do it all over again.

---

To check for a sprite coincidence from this address you will need to type in the following code in your programs.

**CALL PEEK(-31877,N):: IF N AND 32 THEN . . .** (sprite coincidence has occurred.)

This code works the same as **CALL COINC(ALL,N):: IF N THEN . . .** (sprite coincidence has occurred).

---

## 0-255 DOUBLE RANDOM NUMBERS

-31808 This is a two byte random number seed that generates two different random integers with values from 0 to 255.

```
100 RANDOMIZE :: CALL PEEK(-  
31808,A,B):: PRINT A,B :: GO  
TO 100
```

You must execute a **RANDOMIZE** statement prior to peeking into this address to obtain a random number.

The following program will demonstrate the speed difference between **RND** and **CALL PEEK** random number generators. When the program starts running it will generate two random numbers using **RND** and relocate your sprite around the screen. After a short time it will generate two random numbers using **CALL PEEK** and you will be able to see how much faster your sprite is relocated.

```
100 CALL CLEAR :: CALL SCREE  
N(7):: CALL MAGNIFY(2):: CAL  
L SPRITE(#1,42,16,100,100)
```

```
110 FOR N=1 TO 50 :: RANDOMI  
ZE :: A=INT(RND*255):: B=INT  
(RND*255):: CALL LOCATE(#1,A  
/2+1,B+1):: NEXT N
```

```
120 FOR N=1 TO 50 :: RANDOMI  
ZE :: CALL PEEK(-31808,A,B):  
: CALL LOCATE(#1,A/2+1,B+1):  
: NEXT N :: GOTO 110
```



- 
- 100 Clears the screen:: Changes the screen color to dark red:: Sets up sprites to a magnification of 2, a single character made up of 64 dots which occupies 4 character positions on the screen:: Now, we will set up sprite number 1 as character number 42, the asterisk, its color will be white and it will be placed at dot row 100 and dot column 100.
- 110 Now, we will loop here 50 times. First we will **RANDOMIZE** to receive a true random number:: This will define **A** as a random integer with a value between 0 and 255:: This will define **B** as a random integer with a value between 0 and 255:: Now, we will relocate our sprite to a new position according to the values of **A** and **B**. **A** has been divided by 2 to keep our sprite between dot row 1 and dot row 129. Since the **RND** number generator can generate zero as a value we have added 1 to **A** and **B** to prevent an error message from being generated:: Increment **N** and do it over again until **N** is greater than 50.
- 120 After we have relocated our sprite 50 times using the **RND** statement we will loop here 50 times:: We must have a **RANDOMIZE** here to obtain random numbers out of this address:: **PEEK** into -31808 and place two different random numbers into **A** and **B**:: Relocate our sprite to a new position according to the values of **A** and **B**:: Increment **N** and do it over again until **N** is greater than 50:: Jump back up to 110 and do it all over again.

Not only is this random number generator considerably faster, it is also very useful for sprites since it returns values that are within proper limits. **CALL PEEK(-31808,A,B):: CALL MOTION (#1,A-128,B-128)** will give you random sprite motion within the range of -128 to 127. As you will see from some of the other examples in this book, it works quite well as a regular random number generator also.

---

---

## *PATTERNS*

---

---

The following three sprite pattern programs are good examples of arithmetic progressions that can be generated in FOR NEXT loops and applied to sprites with your Texas Instruments 99/4 or 99/4A computer. Sprite pattern 1 requires complete program control to be repetitive but sprite patterns 2 and 3 are self repetitive and they will allow your computer to continue on with the rest of your program once they are generated.

---

## PATTERN 1

```
100 CALL SCREEN(2):: A=1

110 FOR N=1 TO 28 :: CALL SP
RITE(#N,61+A,16,N*6,128+65*A
,0,31*A):: CALL SOUND(-250,3
90+10*A*N,9):: NEXT N :: A=-
A :: GOTO 110
```

100 Changes the screen color to black:: Sets A equal to 1 for later use.

110 Now, we will loop here 28 times and generate 28 sprites:: The key factor to this pattern is the value of A. The first time we generate 28 sprites A will equal 1 then we will change A so that it equals -1 and regenerate our 28 sprites. This program will alternate the value of A each time we have regenerated 28 sprites between -1 and 1. So when A equals 1 we will be generating sprite numbers 1 thru 28 as character 62 and their color will be white. The sprites will be generated on progressive dot rows 6,12,18,24, on thru 168. They will all be generated at dot column 193, ( $128+65*1=193$ ) and have a zero row or vertical velocity. Their column or horizontal velocity will be 31 ( $31*1=31$ ), which will move them to the right:: Now, we will generate sounds in ascending frequencies when A is equal to 1 and descending frequencies when A is equal to -1:: Increment N until N is greater than 28:: Change the sign of A from 1 to -1 or from -1 to 1:: Jump back up to line 110 and regenerate our 28 sprites using the new value of A.

Now let's look at our sprites when A is equal to -1. We will be generating sprite numbers 1 thru 28 as character 60 and their color will be white. The sprites will be generated on the same progressive dot rows but now, we will be generating them all on dot column 63, ( $128+65*-1=63$ ) with a zero row velocity. Their column or horizontal velocity will be -31 which will move them to the left. You can change the column velocity in units of 12 ie., 19,31,43,55 and change the angle of the pattern on the screen.

---

## PATTERN 2

```
100 CALL SCREEN(2):: FOR N=1  
    TO 28 :: CALL SPRITE(#N,42,  
16,N*6,N*2,0,N*4):: NEXT N  
  
110 GOTO 110
```

- 100 Unlike sprite pattern 1 this program will only execute this line once. First, we will change the screen color to black:: Then we will loop here 28 times and generate 28 sprites:: As our loop increments we will be generating sprite numbers 1 thru 28 as character number 42, the asterisk, and their color will be white. They will be generated on progressive dot rows 6,12,18,24, thru 168 and on progressive dot columns 2,4,6,8,10, thru 56. They will all have a zero row velocity, but, we will be generating progressive column velocities of 4,8,12,16,20, thru 112:: Increment **N** until **N** is greater than 28.
- 110 This statement keeps our program running so we can view the sprite pattern.

---

## PATTERN 3

```
100 CALL SCREEN(2):: A=1 ::  
FOR N=1 TO 28 :: CALL SPRITE  
(#N,42,16,N*6,ABS(N+249*(A<1  
)), -6,N*2*A):: A=-A :: NEXT  
N
```

```
110 GOTO 110
```

100 Like sprite pattern 2, this program will only execute this line once. First we will change the screen color to black:: Now, we set A equal to 1:: Then we will loop here and generate 28 sprites:: Once again the value of A is the key factor in this pattern. A will equal 1 for all the odd numbered sprites (1,3,5,7, etc.) and it will equal -1 for all the even numbered sprites (2,4,6,8, etc.). As our loop increments we will be generating sprite numbers 1 thru 28 as character number 42 and their color will be white. They will be generated on progressive dot rows 6,12,18, 24, thru 128. The odd numbered sprites will be generated on progressive dot columns 1,3,5,7, thru 27. Since A is not less than 1 this test will return zero and our equation will appear as  $ABS(N+250*(0)) = ABS(N+0)$ , for the odd numbered sprites. The even numbered sprites will be generated on progressive dot columns 248,246,244, thru 222. Since A is less than 1 this test will return -1 and our equation will appear as  $ABS(N+250*(-1)) = ABS(N-250)$ . All 28 sprites will have a row velocity of -6 which will move them up the screen. The odd numbered sprites will have progressive column velocities of 2,6,10,14, thru 54 and the even numbered sprites will have progressive column velocities of -4,-8,-12,-16, thru -56:: Here is where we alternate the value of A after each sprite is generated between 1 and -1 to correlate with the odd and even numbered sprites:: Increment N until N is greater than 28.

110 Once again, this statement keeps our program running so we can view the sprite pattern.

Again, we see the power of TI computers come shining through. What other home computer can accomplish all this in one line of code and keep running without any further program control?

The two programs in this section will demonstrate a method of making one sprite chase another. There are only two program statements required to set one sprite in motion towards another one. The first one is **CALL POSITION**, which will return the position of both sprites. The second one is **CALL MOTION**, with a simple formula in the row and column velocity parameters. The first program will generate two sprites on the screen and then randomly relocate sprite #1 to different positions. It will then find the position of sprites #1 and #2 and place sprite #2 in motion towards sprite #1. In the second program you will be able to move sprite #1 around with the number 1 joystick and have sprite #2 chase after you.

## CHASE 1

```
100 CALL CLEAR :: CALL SCREE
N(5):: CALL MAGNIFY(2):: CAL
L SPRITE(#1,77,16,100,100,#2
,71,16,30,30)
```

```
110 RANDOMIZE :: CALL PEEK(-
31808,Y,X):: CALL SOUND(60,1
000-Y*3,4):: CALL LOCATE(#1,
Y/2+1,X+1)
```

```
120 CALL POSITION(#2,R,C,#1,
Y,X):: CALL MOTION(#2,(Y-R)*
.49,(X-C)*.49):: CALL SOUND(
300,880-Y*3,15):: GOTO 110
```

- 100 Clears the screen:: Changes the screen color to dark blue:: Sets up the sprites to a magnification of 2:: Sets up sprite #1 as character 77, the M, its color will be white and it will be placed at dot row 100 and dot column 100. This statement also sets up sprite #2 as character 71, the G, its color will be white and it will be placed at dot row 30 and dot column 30.

- 
- 110 Now that everything is set up our program will only loop thru lines 110 and 120.: Since we are using the new random number generator we must execute a **RANDOMIZE** statement prior to peeking into this address.: **PEEK** into -31808 and place two different random integers into Y and X.: Now, we will generate a sound that varies in frequency according to the value of Y. Since Y can be any integer from 0 to 255 our frequencies will vary from 1000 to 235.: Now, we will relocate our sprite to a new position according to the values of Y and X.
- 120 The first thing we will do on this line is find the position of sprites #2 and #1 and place these values into R and C for sprite#2 and into Y and X for sprite #1.: Now, we will set sprite #2 into motion towards sprite #1. The formula that places one sprite, at a given location, in motion towards another location is (DOT ROW TO minus DOT ROW FROM) \*.49, (DOT COLUMN TO minus DOT COLUMN FROM) \*.49. The .49 value is necessary to keep the resultant velocities within proper limits. When our sprites are allowed to be at any possible screen location, the highest velocity obtainable will be 125,  $(256-1)*.49=124.95$ . If your sprites are kept within a confined area such as dot rows 17 thru 185 and dot columns 17 thru 185, then you can adjust this value to obtain maximum velocities within the shorter distances  $(185-17)=168$ ,  $127/168=.7559$ ,  $.75*168=126$ . With this simple formula in our **CALL MOTION** statement the greater the distance the higher the velocity, and the shorter the distance the lower the velocity, but, the time necessary to transverse the distance will remain constant, relative to our final multiplier value.: Now that our sprite is in motion we will generate another sound that varies in frequency according to the value of Y. This sound duration will be used as a delay timer since the duration in the **CALL SOUND** statement in line 110 is a positive value. If you would like to see the effect without a delay change the duration in line 110 to -60.: Now, we will jump back up to line 110 and do it all over again.

---

## CHASE 2

```
100 CALL CLEAR :: CALL SCREE
N(5):: CALL MAGNIFY(2):: CAL
L SPRITE(#1,77,16,100,100,#2
,71,16,30,30)

110 CALL JOYST(1,X,Y):: CALL
MOTION(#1,4*-Y,4*X):: CALL
POSITION(#2,R,C,#1,Y,X):: CA
LL MOTION(#2,(Y-R)*.49,(X-C)
*.49):: CALL COINC(ALL,A)

120 IF A THEN CALL SOUND(-10
0,-2,8):: GOTO 110 ELSE CALL
SOUND(-150,630-R-C,15):: GO
TO 110
```

This program is very similar to Sprite Chase 1 except in this program you are able to move the M sprite around with the number 1 joystick.

### PROGRAM

- 100 Clears the screen:: Changes the screen color to dark blue:: Sets up the sprites to a magnification of 2 :: Sets up sprite 1 as character 77, the M, its color will be white and it will be placed at dot row 100 and dot column 100. This statement also sets up sprite number 2 as character 71, the G, its color will be white and it will be placed at dot row 30 and dot column 30.
- 110 Now that everything is set up our program will only loop thru lines 110 and 120. First, we will look for a joystick input and place -4,0 or 4 into Y and/or X:: Then we will place the number 1 sprite into motion according to the values of Y and X:: Now, we will find the position for sprite numbers 2 and 1 and place these values into R and C and Y and X, respectively:: Now, we will set sprite #2 into motion towards sprite #1, by using our (TO-FROM)\*.49 formula:: After the sprite has been placed in motion we will check for a sprite coincidence and place -1 in A if there is a coincidence or 0 in A if not.



---

120 The IF A statement is the same as IF A<>0. So, if there is a coincidence then this statement will be true and we will generate a type 2 periodic noise sound:: and then jump back up to line 110 to look for a new joystick input ELSE if there was not a coincidence then we will generate a sound that varies in frequency according to the position of sprite #2:: and then we will jump back up to line 110 and do it all over again.

---

---

## SHOOTING

---

---

The example program in this section will show you how to shoot sprites without missing a coincidence. As you type in this program you will notice the CALL COINC subprogram was not used. The theory of operation behind this program is similar to the sprite chase programs in that the  $(TO-FROM) \times .49$  formula was used to set our sprite in motion.

```
100 CALL CLEAR :: CALL SCREE
N(2):: CALL CHAR(46,"0000001
818"):: CALL SPRITE(#2,94,16
,180,1,0,5)

110 FOR N=0 TO 25 :: RANDOMI
ZE :: CALL PEEK(-31808,Y,X):
: CALL SPRITE(#3,65+N,16,Y/2
+1,X+1):: CALL SOUND(-60,660
,8)

120 CALL POSITION(#3,Y,X,#2,
R,C):: CALL SPRITE(#1,46,16,
R,C,(Y-R)*.49,(X-C)*.49):: C
ALL SOUND(476,-3,14)

130 CALL SOUND(120,110,6)::
CALL DELSPRITE(#1):: CALL PA
TTERN(#3,35):: CALL SOUND(10
0,220,6):: NEXT N :: GOTO 11
0
```

*Note: For version 100 extended basic  
change the CALL SOUND statement  
in line 120 to CALL SOUND (425,-3,14).*

- 100 Clears the screen:: Changes the screen color to black:: Sets up character number 46 as small dot, similar to the period character except that the dots in the middle of the character have been turned on:: Sets up sprite #2 as character number 94, the ^ character, its color will be white and it will be placed at dot row 180 and at dot column 1. It will have a zero row or vertical velocity and a column or horizontal velocity of 5 which will move it from left to right across the bottom of the screen.

- 
- 110 Now, we will loop thru lines 110, 120 and 130, 26 times. This loop will randomly place a letter of the alphabet on the screen, find its position, shoot it and then generate the next letter. The first thing we will do in this loop since we are using the new random number generator is execute a **RANDOMIZE** statement:: Next, we will **PEEK** into -31808 and place two different random integers into Y and X:: Now, we will generate sprite number 3 as an alphabet character, A thru Z depending upon the value of N. Its color will be white and it will be randomly placed on the screen according to the values of Y and X. Since we left out the row and column velocity parameters it will be stationary:: Now, we will generate a beep sound.
- 120 Since sprite #3 has been randomly located on the screen and sprite #2 is in motion across the bottom of the screen, we will now find their current positions and place these values into Y and X for sprite #3 and into R and C for sprite #2:: Then we will set up sprite #1 as our dot character, its color will be white and it will be placed at the same dot row and dot column as sprite #2. We will then place it in motion towards sprite #3 by using our  $(TO-FROM)*.49$  formula. Back in the documentation for the Sprite Chase 1 program I stated that the greater the distance between sprites the higher the velocity and the shorter the distance the lower the velocity, but the time needed to transverse the distance would remain constant. In this example the sprite will take approximately 1/2 second to travel from its origin to sprite #3, the alphabet character:: This **CALL SOUND** statement in conjunction with the first **CALL SOUND** statement in line 130 will set up the necessary delay before we delete sprite #1. If you were to change the duration from 476 to 376 your sprite would be deleted prior to reaching sprite #3. On the other hand, if you were to change the duration to 576 then your sprite would pass over sprite #3 and be deleted to late for the right visual effect. So by adjusting the duration in this **CALL SOUND** statement we can obtain the exact amount of delay needed to allow our sprite to just reach sprite #3.

---

130 This CALL SOUND statement must have a positive value duration, in order to keep the previous CALL SOUND statement on for its full duration:: After our CALL SOUND delay is over we will delete sprite #1:: and then we will change the pattern of sprite #3 into the # sign. :: This CALL SOUND statement in conjunction with the first CALL SOUND statement on this line determines the length of time our sprite will appear as # sign before we increment **N** and generate another alphabet character:: Increment **N** until **N** is greater than 25:: When **N** is greater than 25 jump back up to the beginning of line 110 and start generating the alphabet characters all over again.

As you noticed, in this example program, we never miss our target. You can change this by adding some additional code to check the distance between the sprites and if your sprites are in range then allow a hit and if it is not in range then delete it short of your target. You might also check to see if it is lined up on the same row or column with a slight tolerance, before allowing a hit. The time delay between generating your sprite and deleting it can be used to display your score or add to counters or to test to see if you should allow a hit or . . . ? So, with a little additional logic added you can have a lot of control and never miss a coincidence.

---

## ADDENDUM

Listed below are the program lines that may be changed to incorporate joysticks in the Shooting program. The IF ABS (X-C) <9 statement in line 115 sets the tolerance to allow a hit or not. To make it harder to hit use a smaller number such as 4, 5 or 6. The actual tolerance = (number-1) \* 2.

ie:  $(9-1) * 2 = 16$  pixels tolerance.

```
100 CALL CLEAR :: CALL SCREE
N(2):: CALL CHAR(46,"0000001
818"): CALL SPRITE(#2,94,16
,180,100)
```

```
115 CALL JOYST(1,X,Y):: CALL
MOTION(#2,0,X*3):: CALL KEY
(1,X,Y):: IF Y=0 THEN 115 EL
SE CALL POSITION(#3,Y,X,#2,R
,C):: IF ABS(X-C)<9 THEN 120
```

```
116 CALL SPRITE(#1,46,16,R,C
,(Y-R)*.49,0):: CALL SOUND(4
76,-3,14):: CALL SOUND(120,1
10,30):: CALL DELSPRITE(#1):
: GOTO 115
```

```
120 CALL SPRITE(#1,46,16,R,C
,(Y-R)*.49,(X-C)*.49):: CALL
SOUND(476,-3,14)
```

In the two example programs in this section, we will combine together some of the items from the previous sections and demonstrate how to make your sprite pick up and put down objects on the screen. In the first example your sprite will be placed in motion according to the input from the number 1 joystick. The program will then check its position and place a red block underneath it. The second program will generate a screen with the alphabet characters randomly placed on it. Then when you move your sprite around with the number 1 joystick you can pick up one letter. The letter your sprite is holding will be displayed at the upper left hand corner of the screen. To get your sprite to put the letter down, you must place it in the proper alphabetical order, in one of the boxes at the top of the screen. If you do not have joysticks, you can insert the code needed from the **CALL KEY** examples in place of the **CALL JOYST** coding.

---

## PICK UP 1

```
100 CALL CLEAR :: CALL SCREE
N(5):: CALL CHAR(35,"FOFOFOF
O"):: CALL SPRITE(#1,35,2,89
,121):: CALL COLOR(1,1,11,2,
7,7)
110 CALL JOYST(1,X,Y):: CALL
MOTION(#1,-Y*2,X*2):: CALL
POSITION(#1,R,X):: IF R>188
THEN R=1-(Y>0)*187 :: CALL L
OCATE(#1,R,X)
120 CALL GCHAR(INT((R+7)/8),
INT((X+7)/8),Y):: IF Y=32 TH
EN CALL SOUND(-90,660,9):: C
ALL HCHAR(INT((R+7)/8),INT((
X+7)/8),40)
130 CALL KEY(1,X,Y):: IF Y T
HEN CALL CLEAR :: GOTO 110 E
LSE 110
```

*Note: For version 100 extended basic  
change the CALL MOTION statement  
in line 110 to CALL MOTION(#1, -Y,X).*

- 100 Clears the screen:: Sets the screen color to dark blue:: Sets up character number 35 as a small block with the dots turned on in the upper left hand corner of the character:: Sets up sprite number 1 as character number 35, its color will be black and it will be placed at dot row 89 and dot column 121, near the middle of the screen. It will be stationary since we have left out the row and column velocity parameters:: Now, we will change the color of set 1 to a transparent foreground on a dark yellow background and the color of set 2 to dark red foreground on a dark red background. Since the space character, number 32 is in set 1 and we have changed the background color to dark yellow, our screen will now be dark yellow with a dark blue border around it. All of the characters in set 2, character numbers 40 thru 47, will now be dark red blocks since we have set their foreground and background colors in dark red.

---

110 Now that everything is set up we will only loop thru lines 110, 120 and 130. First, we will look for a joystick input and place -4,0 or 4 into X and/or Y.: Now, we will place our sprite into motion according to the values of X and Y. So our sprite will be stationary when the joystick is centered or it will move at a velocity of + or -8 depending upon which direction the stick is pushed in.: Now, we will find the position of the number 1 sprite on the screen. We can reuse X but, we need to retain the value of Y for the next statement so we will place the position values into R and X.: This statement is the same as **IF R IS GREATER THAN 188 THEN IF Y IS GREATER THAN 0 (or the stick is pushed up), THEN R=188:: Relocate our sprite to the bottom of the screen, ELSE IF Y IS NOT GREATER THAN 0 (or the stick is pushed down), THEN R=1:: Relocate our sprite to the top of the screen, ELSE IF R IS NOT GREATER THAN 188 THEN LEAVE R UNCHANGED** and do not relocate our sprite. Here is how it works, If R is less than or equal to 188 we will jump down to the next line. If R is greater than 188 then our sprite has moved off the top or bottom of the screen according to the direction of the joystick. Since our next statement is **CALL GCHAR** and our sprite is off the screen, the computer will generate an error message, to prevent this we will relocate our sprite back onto the screen. When we push the joystick down and go off the bottom of the screen we want to wrap around to the top of the screen and when we go off the top we want to wrap around the bottom. So, when the stick is pushed down and our sprite goes off the bottom of the screen this formula will equate out to  $R=1-(0)*187$  which equals  $R=1-0$ . When the stick is pushed up and our sprite goes off the top of the screen this formula will equate out to  $R=1-(-1)*187$  which equals  $R=1+187$ .: We will then relocate our sprite to the new dot row.



- 
- 120 First we will get the graphics character below our sprite using our dot row/column to graphics row/column conversion formulas. Since we are done with Y we will reuse it and place the character value into Y:: Now, we will check to see if it is character 32, the space character, and if it is we will generate a beep sound:: and then we will place a red block, character number 40, on the screen using our conversion formulas for the row and column position. If the character was not equal to 32, we would have jumped down to line 130.
- 130 Now, we will look for a key press on the left hand side of the keyboard or the number one joystick fire button. Since we are done with X and Y we will reuse them in the CALL KEY subprogram:: This statement is the same as IF Y <> 0. So, if you press the fire button or any key on the left hand side of the keyboard, Y will equal 1, or -1 if you hold it down, and the screen will clear:: Then we will jump back up to line 110 and start all over again. If Y equaled 0 we would jump back up to line 110 to do it all over again without clearing the screen.

---

## PICK UP 2

```
100 CALL CLEAR :: CALL COLOR  
(2,7,7):: CALL SCREEN(11)::  
CALL CHAR(33,"80808080808080  
FF1C5C487F193C26620747E2FFEF  
0E1A33")
```

```
110 CALL HCHAR(24,1,40,64)::  
CALL VCHAR(1,31,40,96):: CA  
LL HCHAR(2,5,33,26):: CALL S  
PRITE(#1,34,2,17,17)
```

```
120 FOR R=1 TO 2 :: FOR C=65  
TO 90 :: RANDOMIZE :: CALL  
PEEK(-31808,X,Y):: CALL HCHA  
R(INT(X/13)+4,INT(Y/10)+4,C)  
:: NEXT C :: NEXT R :: R,C=3
```

```
130 CALL JOYST(1,X,Y):: X=SG  
N(X):: Y=-SGN(Y):: CALL GCHA  
R(R+Y,X+C,CH):: IF CH=40 THE  
N CALL SOUND(-60,110,9):: GO  
TO 130 ELSE C=C+X :: R=R+Y
```

```
140 CALL LOCATE(#1,R*8-7,C*8  
-7):: IF CH=32 THEN 130
```

```
150 IF H=0 AND R>2 THEN CALL  
SOUND(-90,440,9):: CALL PAT  
TERN(#1,35):: H=CH :: CALL H  
CHAR(R,C,32):: CALL HCHAR(2,  
3,CH):: GOTO 130
```

```
160 IF H AND R=2 AND C=H-60  
THEN CALL SOUND(-90,660,9)::  
CALL HCHAR(R,C,H):: CALL PA  
TTERN(#1,34):: H=0 :: CALL H  
CHAR(2,3,32):: GOTO 130
```

---

```
170 IF H THEN CALL SOUND(-90
,-3,9):: GOTO 130 ELSE 130
```

- 100 Clears the screen:: Changes the color of set 2 to a dark red foreground on a dark red background. All the characters in set 2, character numbers 40 thru 47 will now be dark red blocks:: Changes the screen color to dark yellow:: Now, we will redefine characters 33,34 and 35. Character 33 will be the boxes across the top of the screen into which we will place the letters of the alphabet. Character 34 will be the shape of a person and character 35 will be the shape of a person holding an object.
- 110 The next two statements will place a wall around the perimeter of the screen. The CALL HCHAR statement will place the top and bottom walls, character number 40, on the screen starting at row 24, column 1, 64 times. Since there are only 32 columns on the screen, repetition numbers 33 thru 64 will wrap around to the top line of the screen:: The CALL VCHAR statement places the left and right walls on the screen starting at row 1, column 31, 96 times. Since there are only 24 rows on the screen this statement will place 2 complete columns, 48 characters, on the right hand edge and then wrap around to the left hand edge of the screen and place two more complete columns. We have now placed the top, bottom, left and right walls on the screen:: Now, we will place 26 boxes, one for each letter of the alphabet, on the screen starting at row 2 and column 5:: The last thing we will do on this line is to set up sprite number 1 as character 34, our person character, its color will be black, and it will be placed at dot row and dot column 17 which is graphics row and column 3.

---

120 In this line we have two loops, one nested inside the other. The nested loop will generate values that correlate with the character numbers of the alphabet. The outside loop will allow us to execute the nested loop twice. Since the alphabet characters are randomly placed on the screen, it is possible for two different letters to end up with the same row and column position which would leave us with an incomplete alphabet. So, with two sets of alphabet characters being generated we can alleviate this problem. First, we will set up the outside loop to loop twice:: Next, we will set up the inside loop to loop 26 times using the alphabet character values:: Since we are using the new random number generator, we must execute a **RANDOMIZE** statement prior to peeking into this address:: Now, we will **PEEK** into -31808 and place 2 different random numbers into **X** and **Y**:: Now, we will place the letter of the alphabet that correlates with the value of **C** at a random location on the screen. Our random number generator will generate values between 0 and 255, so we will need to convert these values into graphic row and column values. The formula in the row parameter of our **CALL HCHAR** will return values from 4 thru 23 and the formula in the column parameter will return values from 4 to 29:: Now, we will increment **C** until **C** is greater than 90:: Then we will increment **R** and start the nested loop all over again. When **R** is greater than 2 we will leave the two loops:: And set **R** and **C** equal to 3, which is the starting graphic row and column position of our sprite person.

- 
- 130 Now that everything is set up, we will start our main program loop. First, we will look for a joystick input and place -4,0 or 4 into X and/or Y:: This function will convert any negative values of X into -1, any positive value of X into 1 and it will leave X unchanged when it equals 0:: Since the CALL JOYST statement returns Y as -4 when the stick is pushed down and 4 when pushed up we will change this around by placing a minus sign before this function. So now, this function will convert any negative value of Y into 1, any positive value of Y into -1 and it will leave Y unchanged when it equals 0:: Now, we will get the character that occupies the space we want to move our sprite to and place its value into CH:: If CH equals 40 then we are trying to move our sprite into a wall, so we will generate a low frequency sound:: and jump back up to the beginning of line 130 to look for a new joystick input. ELSE we will add the value of X to C:: And the value of Y into R and allow our sprite to move to this new location.
- 140 Now, we will move our sprite to its new location. Since the R and C values represent graphic row and column positions, we will change them into dot row and column values with our conversion formula:: Then we will check to see if we moved our sprite to a space character, number 32, if we did then we will jump back up to line 130 to look for another joystick input.
- 150 If CH did not equal 32, then since we have already tested for a wall character, number 40, it must either equal an alphabet character, numbers 65 thru 90, or a box at row 2 of the screen, character number 33. H is the variable that will tell us which letter of the alphabet our sprite is holding. So if H equals 0, then our sprite is not holding a letter. To prevent our sprite from picking up the letters in our boxes on row 2 we will test to make sure it is not on row 2. So, if it is not holding a letter and it is not on row 2 then we will generate a beep sound:: Change the pattern of our person sprite into the shape of a person holding an object:: Then we will set H equal to the character value of the letter that is at the same screen location as our sprite:: Next, we will place a space character at this location to delete this letter from the screen:: Then, we will display the letter our sprite is holding, at the top left hand corner of the screen:: And finally, we will jump back up to line 130 to look for a new joystick input.

- 
- 160 If our sprite was holding a letter or if it was on row 2, then the test in line 150 would be false and we would jump down to this line. If H is the same as IF H<>0 so if H does not equal 0 it must equal the character value of one of the letters of the alphabet. So, when our sprite is holding a letter and it is on row 2, then when it is lined up with the box that correlates with the proper alphabetical position of the letter it is holding, we will put the letter down. Since we placed 26 boxes on row 2, starting at column 5, we will subtract 60 from the value of the letter our sprite is holding and test to see if we are at the proper column. The A can only be placed at column 5, since the character value of A equals 65 and 65-60 equals 5, the B will be placed at column 6 and so on. So, if our sprite is holding a letter and it is on row 2 and at the proper column, we will generate a beep sound:: And place the letter on the screen at row 2 and at the column that correlates with the value of C:: Then, we will change the pattern of our sprite back into a person that is not holding anything:: Next, we will set H equal to 0 to indicate that our sprite is not holding a letter:: Then, we will delete the letter from the upper left hand corner of the screen:: Finally, we will jump back up to line 130 to look for a new joystick input.
- 170 If the tests in line 150 and line 160 were false, then if our sprite is holding a letter and it is standing on another letter that is not on row 2, or it is holding a letter and is on row 2, but it is not at the proper column position, we will generate a type 3 periodic noise sound:: and jump back up to line 130 to look for another joystick input. We will execute the ELSE statement when our sprite is not holding a letter, but it is on row 2 which will jump us back up to line 130 to look for another joystick input.

This is just the start of an educational program, from here you could add better reward sounds when the letter is put down. You could also keep track of the letters that were placed in the proper box on the first try and display a percentage score. You might also want to have the program check to see when the boxes are full, so, the game can be started over or ...? From here it is up to you, so let your imagination be your guide, and take full advantage of the power your TI computer has in Extended Basic.

---

## NOTES

---

---

## *DOTS AND TRAIL*

---

---

In this section, the first program will set up a sprite that eats dots as you move it around with the number 1 joystick. It will also lay down a blue wall trail behind your sprite, open and close its mouth and change its pattern to correlate with the direction it is headed in. The second program will generate a maze on the screen and as you move your sprite around, it will paint the floor blue. Your sprite will not be allowed to move around on the blue areas, so to paint all of the floor you will have to plan your moves very carefully. Even though, at first it may seem impossible there is a solution to this puzzle. If you get stuck and want to start over, you can press the fire button or the Q key and all of the blue areas will be erased from the screen. To make it easier, you are allowed to move diagonally with either the number 1 joystick or the arrow keys E,S,D and X and the diagonal keys W,R,Z and C.



---

## DOTS

```
100 CALL CLEAR :: CALL SCREE
N(11):: CALL CHAR(33,"000000
006060",96,"AAFFFEDEFBEBBBEBF
AAFFFE8F8E9B9E9FAAFFDEFFFEF7
OEFFAAFFDEFFE60706FF")
```

```
110 CALL CHAR(100,"BFBE BBBED
FFEFFAA9F9E9B8E8FFEFFAA55FF7
BFF7FEF70FF55FF7BFF67E060FF"
):: CALL COLOR(1,15,2,2,5,5)
```

```
120 CALL HCHAR(24,1,40,64)::
CALL VCHAR(1,31,40,96):: P=
102 :: CALL SPRITE(#1,P,16,1
7,17)
```

```
130 FOR R=1 TO 50 :: RANDOMI
ZE :: CALL PEEK(-31808,X,Y):
: CALL HCHAR(INT(X/12)+2,INT
(Y/10)+4,33):: NEXT R :: R,C
=3
```

```
140 CALL JOYST(1,X,Y):: IF X
AND Y THEN 140 ELSE IF X TH
EN P=100+X/2 :: X=SGN(X)ELSE
IF Y THEN P=98+Y/2 :: Y=-SG
N(Y)
```

```
150 CALL GCHAR(R+Y,X+C,CH)::
IF CH=40 THEN CALL SOUND(-6
0,110,9):: GOTO 140 ELSE C=C
+X :: R=R+Y
```

---

```
160 CALL PATTERN(#1,P):: CAL
L LOCATE(#1,R*8-7,C*8-7):: I
F CH=33 THEN CALL SOUND(-100
,-7,6):: CALL HCHAR(R,C,32)
```

```
170 CALL PATTERN(#1,P+1):: I
F X OR Y THEN CALL SOUND(-90
,660,9):: CALL HCHAR(R-Y,C-X
,40):: GOTO 140 ELSE 140
```

- 100 Clears the screen:: Changes the screen color to dark yellow:: redefines character numbers 33,96,97,98 and 99. Character number 33 will be the dots our sprite eats. This character was defined so that the dots would line up with our sprites mouth. Character number 96 is our sprite facing down with its mouth closed, 97 is our sprite facing down with its mouth open. Number 98 faces left with its mouth closed, 99 faces left with its mouth open.
- 110 This will redefine character numbers 100,101,102 and 103. Character number 100 is our sprite facing up with its mouth closed, 101 is our sprite facing up with its mouth open. Number 102 faces right with its mouth closed and 103 faces right with its mouth open:: Now, we will change the color of sets 1 and 2. Set 1, character numbers 32 thru 39 contains the space character and our dots. The space character will now be black and our dots will be gray on a black background. Set 2, character numbers 40 thru 47 contains our walls and the trail our sprite leaves. All the characters in this set will now be dark blue blocks.

- 
- 120 The next two statements will place a dark blue wall around the perimeter of the screen. The **CALL HCHAR** statement will place the top and bottom walls, character number 40, on the screen starting at row 24, column 1, 64 times. Repetition numbers 33 thru 64 will wrap around and be placed on the top row since there are only 32 columns on the screen. The **CALL VCHAR** statement places the left and right walls on the screen starting at row 1, column 31, 96 times. Since there are only 24 rows on the screen, this statement will place 2 complete columns, 48 characters, on the right hand edge and then wrap around to the left hand edge and place two more complete columns. Now, we will set **P** equal to 102, which is the starting pattern, character number, for our sprite. Then, we will set up sprite number 1 as character 102, which is the right facing sprite with its mouth closed. Its color will be white and it will be placed at dot row and dot column 17, which is graphics row and column 3.
- 130 Now, we will set up to loop here 50 times and randomly place our dots on the screen. Since we are using the new random number generator, we must execute a **RANDOMIZE** statement prior to peeking into this address. Then we will **PEEK** into -31808 and place 2 different numbers into **X** and **Y**. Since our random number generator places values from 0 thru 255 into **X** and **Y** we will need to convert these values into valid graphic row and column values. The formula in the row parameter of our **CALL HCHAR** statement will return values from 2 thru 23 and the formula in the column parameter will return values from 4 thru 29. Then we will increment **R** and generate another dot on the screen until **R** is greater than 50. Since we are done with **R** we will reuse it and set **R** and **C** equal to 3, which is the starting graphic row and column positions of our sprite.

- 
- 140 Now that everything is set up, we will start our main loop which will loop thru lines 140,150,160 and 170. First, we will look for a joystick input and place -4,0, or 4 into X and/or Y:: This statement is the same as IF X<>0 AND Y<>0 and will only be true when you have moved the joystick into a diagonal position. If this is true, then, we will jump back to the beginning of line 140 to look for a new joystick input. ELSE IF X is the same as IF X <>0 and this statement will be true when the stick is moved to the right or the left. If the stick is moved to the left X will equal -4 and P will equal 98,  $P=100+(-4)/2$  which equals  $P=100-2$ . If the stick is moved to the right then X will equal 4 and P will equal 102,  $P=100+4/2$  which equals  $P=100+2$ . P is the variable that carries the proper sprite pattern, character number, down to lines 160 and 170 to change the direction our sprites are facing in, according to joystick movement:: We will now change the value of X to -1 when X is negative or to 1 when X is positive. So, if we have moved the stick right or left, at this point we would jump down to line 150 to continue execution. If the first and second tests were false then we will check to see if the stick was pushed up or down. Once again, the IF Y statement is the same as IF Y<>0 will only be true if the stick is pushed up or down. If the stick is pushed up, then Y will equal 4 and P will equal 100,  $P=98+4/2$  which equals  $P=98+2$ . If the stick is pushed down, then Y will equal -4 and P will equal 96,  $P=98+(-4)/2$  which equals  $P=98-2$ :: Then we will change the value of Y to -1 when Y is positive or to 1 when Y is negative. If the joystick is not moved, then X and Y will equal 0, so all three tests will be false and our sprite will continue to face in the direction it was placed in from the last joystick input.
- 150 This statement will make the computer get the screen character at the location we want to move our sprite to and place its value into CH. So, if we push the stick down then the value of the character directly below our sprite will be placed into CH. The same is true for the other possible joystick positions:: If CH equals 40 then we are trying to move our sprite into a blue wall, so we will generate a low frequency sound:: and jump back up to line 140 to look for a new joystick input. If CH did not equal 40, then we can move our sprite to this new location so we will add the value of X to C:: and the value of Y to R.

- 
- 160 This statement will set up our sprite with its mouth closed and facing in the proper direction according to value of P we obtained from line 140.: Next, we will place our sprite in its new location if we have moved the joystick and changed the value of R or C. Since R and C contain values that correlate with graphic row and column positions, we will change them into dot row and column values by using our conversion formula.: Now, we will test to see if we have moved our sprite on top of a dot, character number 33. If we have, then we will generate a type 7 white noise sound.: and place a space character on the screen at this location to delete the dot. If CH did not equal 33, then we would have jumped down to line 170 and continue program execution from there.
- 170 This statement will set up our sprite with its mouth open, P+1 equals the character number for the open mouth sprite. Since the value of P will not have been changed since the last CALL PATTERN statement, our sprite will still be facing in the same direction. The first CALL PATTERN statement sets up our sprite to face in the proper direction, according to the joystick input, with its mouth closed and the second CALL PATTERN statement opens its mouth.: Now, we will test to see if the joystick has been moved out of center position. The IF X OR Y statement is the same as IF X<>0 OR Y<>0. So, if we have moved the joystick then X or Y will equal -1 or 1 and we will generate a beep sound.: Then, we will place a blue block on the screen in the position that we just moved our sprite from. So, with this test we will only generate beep sounds and place blue blocks on the screen when our sprite is moved.: Now, we will jump back up to line 140 to look for a new joystick input. If we did not move the joystick, then we will jump back up to line 140 without generating a beep sound or placing a blue block on the screen.

---

Once again, this is just the start of a game program that you can add any number of items to. You might want to incorporate part of the alphabet pick up program in this program. You could replace the code in line 130 of this program with the code in line 120 of the Pick up 2 program. Then you could change the code in line 160 to . . . IF CH=65+N THEN CALL SOUND (-100,-7,6):: CALL HCHAR(R,C,32)::N=N+1. This would set up the program so that you will have to eat letters in alphabetical order, but you will have to plan your moves very carefully, since your sprite lays down a wall whenever it moves. Then with a simple IF N=26 test you could restart the game after one complete alphabet has been eaten. It is hard to believe that this program only uses 1 K, 1000 bytes of memory. We have all those nice subprograms that are built into our ROM to thank for our easy, but, powerful programming capabilities. So, I will leave it up to you to see what enhancements and perils you can add to this program with all that RAM you have left.

---

## NOTES

---

## MAZE PUZZLE

```
100 CALL CLEAR :: CALL SCREE
N(2):: CALL CHAR(35,"FFFFFFF
FFFFFFF",42,"1C5C487F193C2
662"):: CALL COLOR(1,7,12,2,
6,6)
```

```
110 A$=" ## ### ## ###
## # # #
## # " ::
CALL HCHAR(24,1,35,64):: CAL
L VCHAR(1,31,35,96)
```

```
120 CALL SPRITE(#1,42,2,177,
17):: DISPLAY AT(2,1): :A$&A
$:A$&A$:A$&A$:A$ :: Y=23 ::
X=3
```

```
130 CALL JOYST(1,C,R):: R=-S
GN(R):: C=SGN(C):: IF R OR C
THEN 150 ELSE CALL KEY(1,C,
R):: IF C=18 THEN 120
```

```
140 R=(C>3 AND C<7)-(C=0 OR
C=15 OR C=14):: C=(C=2 OR C=
4 OR C=15)-(C=3 OR C=6 OR C=
14)
```

```
150 CALL GCHAR(Y+R,X+C,CH)::
IF CH>34 THEN 130
```

```
160 Y=Y+R :: X=X+C :: CALL S
OUND(-90,-2,4):: CALL LOCATE
(#1,Y*8-7,X*8-7):: CALL HCHA
R(Y,X,40):: GOTO 130
```



- 
- 100 Clears the screen:: Changes screen color to black:: Defines character number 35 as a block with all the dots on and defines character number 42 into the shape of a person:: Changes color of character set 1, character numbers 32 thru 39, to a dark red foreground on a light yellow background. All space characters will now be yellow and our blocks, character number 35, will be dark red. Also changes color of character set 2, character numbers 40 thru 47, to a light blue foreground on a light blue background. All the characters in set 2 will now be light blue blocks.
- 110 Defines A\$ as spaces and # signs. The # sign is character number 35 and we have already defined it as a block. When A\$ is displayed on the screen, it will look like a maze with red walls and a yellow floor:: Now, we will place a wall around our maze which we will not allow our person sprite to go through. The CALL HCHAR statement places our blocks, character 35, starting at row 24 and column 1, 64 times. Since there are only 32 columns on the screen, repetition numbers 33 thru 64 will wrap around to the top of the screen. This forms the top and bottom walls of our maze area:: The CALL VCHAR statements starts at row 1, column 31 and places 96 blocks on the sides. 48 blocks or 2 full columns on the right hand edge of the screen, then it wraps around to the left hand edge of the screen and places another 48 blocks or 2 more full columns. So, with one CALL HCHAR and one CALL VCHAR statement we have placed the top, bottom, left side and right side walls on the screen. We have also left rows 2 thru 23 and columns 3 thru 30 empty. Columns 3 thru 30 are the same as display columns 1 thru 28, or text columns.
- 120 Sets up sprite number 1 as our person, character number 42, sets the sprite color as black and places it at dot row 177 and dot column 17. Remember, our formula  $R*8-7=DR$ ,  $23*8-7=177$ ,  $3*8-7=17$  so, our sprite is now sitting at row 23, column 3:: Now, we will display our maze pattern that we defined in line 110. First, we will display a blank line then we will display two A\$ put together then we will go down to the next line and display two more A\$ put together, then we will go down to the next line and display two more A\$ put together and finally, we will go down to the next line and display one more A\$. Since each A\$ is 71 characters, it will use up a little over 2 1/2 lines:: Now, we set Y=23 and X=3 which is the starting graphic row and column of our sprite.

---

130 Now that everything is set up, this is where the game loop starts, so, from now until the game is finished or until we clear the screen the program will only loop thru lines 130, 140, 150 and 160. The first thing that the computer will do on this line is look for a joystick input. If the number 1 joystick is moved then R or C or both will contain -4 or 4 depending on how the stick is moved.: The R variable will contain the row commands of the CALL JOYST statement. Since the R value is negative when the joystick is moved down and positive when it is moved up, we need to change this around. By placing a negative function before the R variable it will now return negative numbers when the stick is pushed up and positive numbers when pushed down. The -SGN(R) function will only return -1,0 or 1 for any value assigned to R. If R is a negative number, this function will return 1. If R is zero it will return 0. If R is positive it will return -1. By using this function, we have converted the -4, 0 or 4 values the CALL JOYST statement returns into 1, 0 or -1.: The C variable will contain the column commands of the CALL JOYST statement. Since the C value returns the proper negative number when the stick is pushed left and a positive negative number when it is pushed right, we do not need to change it around like the R value. Once again, we use the SGN(C) function to return -1, 0 or 1 for use in moving our sprite in line 160.: Now, we will check to see if the joystick was moved. The IF R OR C statement is the same as IF R<>0 OR C<>0 except that it uses less bytes and it executes a little faster. If the joystick was moved R or C or both will now have a value of -1 or 1 depending on where the stick was moved to. If the stick was moved, then the program jumps down to line 150 to continue execution. If the stick was not moved, then the program executes the CALL KEY(1,C,R) statement. This function looks for any key press on the left hand side of the console or for the fire button on the number 1 joystick.: If we have pressed the fire button or the Q key then the program jumps up to line 120, which will start the game over by relocating our sprite to its starting position and by redisplaying all of our A\$'s. This clears all the blue areas off the screen and continues the program from there.

- 
- 140 This line is a good example of the use of logical and relational expressions without using IF THEN ELSE statements. If we did not move the joystick, then we looked for a key press. Since we do not need the status of the key press we will reuse the R variable. The first statement group in this line is the same as IF C=4 OR C=5 OR C=6 THEN R=-1 ELSE IF C=0 OR C=15 OR C=14 THEN R=1 ELSE R=0. So if C did not equal any of these keys then R=0. The second statement group is the same as IF C=2 OR C=4 OR C=15 THEN C=-1 ELSE IF C=3 OR C=6 OR C=14 THEN C=1 ELSE C=0. Since the computer completes all these tests before assigning a value of -1, 0 or 1 to C we can reuse the C variable to save bytes and speed up our program. These two statement groups plus the CALL KEY statement have now given us all the information we need to use the four arrow keys (ESDX) plus the diagonal keys (WRZC) to move our sprite around the maze. When the program is running you will notice that the joystick moves the sprite around much faster since it has less code to execute.
- 150 This is the line that controls where our sprite can and cannot be moved to. When the program starts up, Y=23 and X=3 and since you probably will not have moved the joystick or pressed a key, R and C will both equal 0. So on the first time thru the loop we get the character at row 23, column 3, beneath our sprite. This character will be a space, number 32, so CH will equal 32. Since CH is not greater than 34 our program will jump down to execute line 160. On our first pass thru the loop line 160 will place a blue square, character number 40, under our sprite. Now, on the second pass thru the loop if we still have not pressed a key or moved the stick then CH will equal 40 so we jump back up to line 130 to start the loop over again. Now let's say on the third time through the loop we press X or move the joystick down, then R will equal 1 and C will equal 0 so  $Y+R=23+1=24$  and  $X+C=3+0=3$ . Now we will get the character at row 24 and column 3 and since there is a wall below our sprite which is character number 35, CH will equal 35 and since 35 is greater than 34 we will jump up to line 130 without executing any portion of line 160. The CALL GCHAR (Y+R,X+C,CH) statement makes the program look in the direction we want to move our sprite. If there is a space, character 32, then we will be allowed to execute line 160 which moves us there. If there is a wall or a blue square, character number 35 or 40, then we will jump to line 130 and our sprite will not be moved.

---

160 In this line, we will add R and C to the values of Y and X, make a beep sound, move our sprite, put a blue square beneath our sprite and then go back to the top of the loop. This line will only be executed if and only if the key pressed or the joystick movement we chose will move our sprite onto a space character, the yellow area. Lets say that the first key pressed is D or that we move the stick to the right. R will equal 0 and C will equal 1 so  $Y=Y+R$  or  $23+0=23$  and  $X=X+C$  or  $3+1=4$ . Y will still equal 23 and X will equal 4.:: Generate a type 2 periodic noise sound.:: Now, we will take our row Y and column X values and relocate our sprite to its new position.  $23*8-7=177$  which is the same dot row we started at.  $4*8-7=25$  which is eight dot columns to the right of where we started.:: Now we will place a blue square, character 40 at row 23, column 4.:: Jump back up to line 130 and do it all over again.

Once you have mastered this maze you might try changing A\$ in line 110 by adding more # in place of space characters you might also change the code in lines 130 and 140 to delete the diagonal moves. You might also add another character like blue squares with foot prints on them and change the code in lines 150 and 160 to allow you to move around on the blue squares, but, you will leave foot prints and have to repaint that section of floor. Once again, we see how powerful the TI 99/4(A) home computer is in extended basic. I doubt that most of the other home computers out there can set up a program such as this in only 7 program lines.

---

## NOTES

This general bar grapher was written to handle a variety of bar graphing needs. The smallest scale that can be displayed is 0 to 200 with each pixel having a value of one. The largest number this bar grapher can handle is 9,999,999,999. In this example program below lines 100 thru 150 are used to set up the information to be transferred to the subprogram GRAPH. Lines 160 thru 270 set up and display the graph. The graph subprogram is complete in itself. It will define its own characters, set its own colors, display the name of the graph, display the scale multiplier as well as the value between dots on the scale. It will then generate a bar graph for 1 to 20 items, afterwhich, it will return control back to your main program.

## GENERAL BAR GRAPHER

```
100 CALL CLEAR :: DIM G(20),  
G$(20)
```

```
110 DATA JAN,187.96,FEB,137.  
84,MAR,199.83,APR,144.77,MAY  
,80.58,JUN,291.94,JUL,166.79  
,AUG,188.14,SEP,190.37
```

```
120 DATA OCT,262.26,NOV,269.  
62,DEC,268.06,AVG,199.01,81,  
186.54,80,171.13,79,143.31,7  
8,121.63,77,107.07,76,96.53
```

```
130 FOR I=1 TO 18 :: READ G$(  
I),G(I):: NEXT I :: G$(0)="GROCERY EXPENSE 82"
```

```
140 CALL GRAPH(G$(),G())
```

```
150 GOTO 150
```

---

```
160 SUB GRAPH(G$( ),G( )):: CA
LL CLEAR :: CALL SCREEN(7)::
  FOR I=1 TO 8 :: CALL COLOR(
I,2,12):: NEXT I :: CALL COL
OR(2,13,4,9,13,4)
```

```
170 CALL CHAR(43,"0101010101
010101",45,"00000000001",62,"
0000000000001",94,"0000000101
0101",96,"00FFFFFFFFFFFFFFFF")
```

```
180 M,C=0 :: C$="-+---+-----+
---+-----+-----+" :: DISPLAY A
T(1,(29-LEN(G$(0)))/2):G$(0)
: : ":" ^>>>>^>>>>^>>>>^>>>
^>>>>^"
```

```
190 FOR I=1 TO 20 :: IF G$(I
)<>" THEN DISPLAY AT(I+4,4)
:C$ :: M=MAX(M,G(I))ELSE 210
```

```
200 NEXT I
```

```
210 C$="8080C0E0F0F8FCFE" ::
  IF M<9801 THEN MX=1 ELSE MX
=10^(LEN(STR$(INT(M)))-3)
```

```
220 M=INT((M-1)/200)+1 :: DI
SPLAY AT(2,1):"SCALE X"&STR$
(MX)&" > ="&STR$(M*8)
```

---

```
230 DISPLAY AT(3,3):USING "0
    ##### "4
0/MX*M,80/MX*M,120/MX*M,160/
MX*M,200/MX*M
```

```
240 FOR I=1 TO I-1 :: DISPLA
Y AT(I+4,1)SIZE(3):G$(I):: I
F G(I)<M*8 THEN 260
```

```
250 CALL HCHAR(I+4,6,96,INT(
G(I)/(M*8))): CALL SPRITE(#
I,96,13,(I+4)*8-7,G(I)/M+33)
:: GOTO 270
```

```
260 IF G(I)>0 THEN CALL CHAR
(97+C,"00"&RPT$(SEG$(C$,INT(
G(I)/M)*2+1,2),7)): CALL SP
RITE(#I,97+C,13,(I+4)*8-7,41
):: C=C+1
```

```
270 NEXT I :: SUBEND
```



- 
- 100 Clears the screen:: This will dimension G and G\$ to handle 21 items. G\$ is used to identify each bar of your graph, such as Jan., Feb., Mar., etc., or food, gasoline, electricity etc. G\$ may be any length from 1 to 255 characters but the GRAPH subprogram will only display the first 3 characters. The G variables are used for the value or amount of the items that you have labeled with G\$. The subprogram GRAPH will convert these values into color bars of various lengths and display them on our grid. When you use the subprogram GRAPH in one of your programs you will need to place the DIM G(20), G\$(20) statement in your program prior to any reference to G or G\$.
- 110 This line contains the data we will use for our example graph. This data statement contains my grocery expenses for the months of January thru September.
- 120 This line contains the balance of the data we will use for our example graph. In this data statement I have my grocery expenses for the months of October thru December and the average monthly expenses for 82, 81, 80, 79, 78 and 77.
- 130 Now we will read all the data and place it into G\$ and G. Since there are 18 items we want to read we will loop here 18 times. The first time thru the loop we will assign JAN to G\$(1) and 187.96 to G(1). The second time thru the loop we will assign FEB to G\$(2) and 137.84 to G(2). We will continue this procedure right on thru the 18th time when we will assign 77 to G\$(18) and 107.07 to G(18) after which we will leave the loop:: Now we will give our graph a title. G\$(0) is the subscripted string that will carry our graph title into our subprogram GRAPH. G\$(0) can be any length from 0, for no title, to 28 characters, 1 full text line. The subprogram GRAPH is set up in such a way that you should only use upper case characters when you define all the G\$. Since we have changed the color of character set 2 to form a grid for our graph, you should not use any of the following characters in any of the G\$'s (\*+,-./).

- 
- 140 This is the line that causes the bar graph to be displayed on your screen. Let's say you have placed the subprogram GRAPH at the bottom, highest line numbers, of one of your existing programs, and that you have placed the DIM G(20),G\$(20) statement at or near the top of your program. Now anytime during your program execution, after you or your program has assigned names and values to G\$ and G, the statement CALL GRAPH(G\$(),G()) is encountered, a complete bar graph will be displayed on the screen.
- 150 After the bar graph has been displayed on the screen the program returns to this line. Since we want a little time to view the graph the GOTO 150 statement will keep the program from ending. When you are done looking at the graph just press shift C or FCTN 4 (clear) to stop the program. If you incorporate the bar grapher in your programs you might want to program in a CALL KEY statement that will look for a key press to continue your program after the CALL GRAPH(G\$(),G()) statement has been executed.
- 160 This is where the subprogram (GRAPH (G\$(),G())) starts. Lines 160 thru line 270 is the total bar graphing subprogram. If you incorporate this subprogram into one of your existing programs you will need to remember these lines so that they are the highest line numbers in your program. If you do renumber this section, make sure you change the line number after the ELSE statement in line 190 and the line number after the THEN statement in line 240 to correspond to the renumbered lines. Now lets see how this program works. First we will clear the screen:: Then we will change the screen color to dark red:: Now we will set up a loop to change the foreground and background colors of character sets 1 thru 8 to black on light yellow:: And the last thing we will do on this line is to change the colors of character sets 2 and 9 to a dark green foreground on a light green background. Character set 2 contains the + and - characters which we will redefine and use to display a grid our graph bars will be on. Character set 9 contains character number 96 which we will redefine and use as our bars.

- 
- 170 In this line we are redefining 5 characters. Character 43, the + character will now be a vertical line. Character 45, the - character will now be a dot. Character 62, the > character will now be a dot. Character 94 the ^ character will now be a short vertical line and character 96 will now be a block with the top pixels off. Characters 43 and 45 are used to display our grid. Characters 63 and 94 are used to indicate our scale dividers and character 96 is used for the bars of the graph.
- 180 First we will make sure M and C are cleared out or equal to zero:: Then we will define C\$ as - and + characters, which will look like dots and lines when the program is running, for later use:: Remember, G\$(0) which is the title of the graph, this is where we will put it on the screen. By using this DISPLAY AT statement with this short formula in the column section of the AT we can center any string on the screen that is less than 27 characters. Then we will display two blank lines followed by the ^ and > characters. These characters have been redefined as dots and short lines and they will be placed under our scale numbers.
- 190 In this loop we are going to do three things:: First we will check to see if the subscripted string has any characters in it. If it does we will continue on, if it does not we will leave the loop. Lets look at our example data, we have loaded 18 of the possible 20, G\$'s which we are using as bar identifiers, we are not counting G\$(0) which is the title string. When I equals 19 and this line checks G\$(19) it will find it is an empty or null string and we will leave the loop, retaining I as being equal to 19 for later use. :: Second, since G\$(1) thru G\$(18) contains characters we will display 18 lines of the grid:: Third, we will check each subscripted G variable from G(1) thru G(18) to find the largest value of G and we will assign this maximum value to M.
- 200 This is the line we jump over if G\$(I) is null to leave the loop.

---

210 Ok, now that we are finished with C\$ we will reuse it and define it with these hexadecimal values for later use. The only reason it was placed here is because there is room for it on this line, we do not need to use it until line 260:: In line 190 we found the largest value our graph is to display and in our example JUN has the highest value so at this point M is equal to 291.94. Since M is less than 9801 we will set MX equal to 1. MX is our scale multiplier, and when we display the scale at the top of the graph there is room for 4 digits to be displayed between the short indicator lines. Since our scale increases in increments of 200, if the highest value our scale will display is greater than 9800 we will end up with more than 4 digits. This is where the scale multiplier MX comes into play. Lets say our highest value is 9980 then M is not less than 9801, and our scale will be 0 to 10,000, so we will execute the ELSE statement. Since there are 5 digits in 10,000 we need to shorten this display down to 3 or 4 digits. When our scale is displayed it will show values between 0 and 1000 and our scale multiplier will be 10 so we will multiply all the scale values shown by 10 to get their real values. For ease in reading the scale this was set up so that our scale multiplier MX will always be a power of 10 such as 1, 10, 100, 10000, etc. The ELSE statement will take care of this by converting the integer of M or 9980 into a string, it will then find the length of the string, in this case it is 4, subtract three from it, leaving us with a value of 1. 10 raised to the power of 1 is 10 so MX will equal 10. This ELSE statement is the section of the graph program that limits us to a maximum value of 9999999999. If you type in PRINT 9999999999 the computer prints 9999999999 but if you type in PRINT 10000000000 the computer prints 1.E+10. Since it has been converted to scientific notation the length of the string M is now 6, it should be 11, so this throws our scale display out by 5 digits and leaves us without a scale. The bar graphs and value between dots will still be accurate, but the scale multiplier and scale are incorrect.

- 
- 220 Now that we have determined what scale will be displayed and what the scale multiplier is, we need to find out what the value of each pixel will be. Since our graph is 200 pixels wide, 25 columns times 8 pixels per column=200, this formula will convert  $M$  from the highest value our graph will display into the value of each pixel for later use. When we run our example program at this point  $M$  will equal 2:: Here is where we display the scale multiplier and the value between the dots on our grid. In our example the scale multiplier is equal to 1 and  $M$ , or each pixel, is equal to 2, and the value between the dots is 16, ( $M \times 8$ ).
- 230 In this line we will display the scale across the top of the grid. The **USING** statement allows us to format our numbers so that they will always be displayed next to the short indicator lines of the grid. it also limits us to a 4 digit display. Since our scale will always be in multiples of 200 to correlate with the pixels, the 5 divisions in the scale will always be relative to the highest scale value. This is how 40, 80, 120, 160 and 200 came to be our base numbers for this display. In our example  $MX$  is 1 and  $M$  is 2 and since the computer always completes division before multiplication our values are computed as  $40/1=40, 40 \times 2=80$  not  $40/2$  which equals 20. After these values are computed they are placed into the corresponding # sign groups and displayed on the screen.

- 
- 240 Ok, now that everything is set up this is where we start displaying our bar titles and bars. Remember back in line 190 we had 18 items to display and when we left that loop  $I$  was equal to 19, here is where we use that value. Since the computer sets up the starting and ending values before it starts the loop, this loop will be from 1 TO  $I-1$  or in our example 19-1 or 18:: This statement will display the first 3 characters of our bar titles starting at row 5 and in the case of our example program ending at row 22:: After the title is displayed the program will check to see if the value, or length of the bar to be displayed is less than the value of one column of pixels. In our example one column of pixels is equal to 16 dollars and the lowest amount we will display is 80.58 so the program will continue on the next line instead of jumping to line 260.
- 250 This is where we display the bars and place the sprite at the end of them for one pixel accuracy. Lets say that this is the first time thru the loop in our example program.  $I$  will equal 1, so starting at row 5 column 6, do not forget that this is a graphics column not a text column, we place character 96, our dark green block, 11 times to form a horizontal bar. The 11 comes from our formula,  $\text{INT}(187.96/(2*8))=11$ :: Since we are still on the first loop thru the program so we will be setting up sprite #1 as a block, character 96, in a dark green color. Then we will place it at graphic row 5,  $(I+4)$  or dot row 33,  $5*8-7=33$ . The value of  $G(I)$  will determine what dot column to place the sprite in. In our example it will be placed in dot column 127.  $187.96/2 + 33=126.98$ , this formula is a condensed form of our graphics to dot column conversion. It started out as  $(G(I)/(M*8))*8-7+41-1$  and was condensed down to  $G(I)/M+33$ , 41 is the dot column where all the bars start and the -1 at the end of the prior formula is to compensate for rounding off:: Now that we have displayed the bar and placed our sprite at the end of the bar we will jump down to line 270 to increment  $I$  and display the next **G\$** and **G**.

---

260 The only time our program will execute this line is when  $G(I)$  is less than  $M*8$ . In our example program  $M*8$  is equal to 16 and our lowest value is 80.58, so this line is never executed. Lets say our lowest value is 12, then when the program encounters the test in line 240 we will jump down to this line. Since we jumped over line 250 we will not have placed any blocks on the screen for this value. The first thing we will do on this line is to test  $G(I)$  to see if it is greater than 0. If  $G(I)$  is a negative number or if it is equal to zero we will leave this line and jump down to line 270. If  $G(I)$  has value greater than 0 we will continue on with the rest of this line. Do not forget that the only time we will use this line is when we need to display less than 1 full block. Now lets get back to  $G(I)=12$ . First we will define a character that has the proper amount of dots turned on to correspond with  $G(I)$ . In our example each pixel equals 2 dollars so when  $G(I)=12$  we will have to define a block with the first 6 columns of pixels or dots turned on. This is where we use that  $C\$$  that was defined in line 210. Lets say that May's value is 12 and that this is the first bar that has a value less than 16. Since we cleared out  $C$  to zero in line 180 we will be redefining character number 97. First we turn off the top row of dots like we did for character 96 in line 170. Next we will take a 2 character segment of  $C\$$  and repeat it 7 times to finish defining character 97. If  $G(I)=12$  then when  $M$  equals  $2 \text{ INT}(G(I)/M)*2+1=13$ . So we will use the 13th and 14th characters of  $C\$$  or  $FC$ . The `CALL CHAR` statement at this point is equivalent to "00FCFCFCFCFCFC", which will give us the 6 columns of dots or pixels we need. Now that our character is defined we will place it on the screen as a sprite. Since we are using May as an example I will equal 5 at this point, so we will set up sprite number 5 as character 97, the one we just defined, in a dark green color. Then we will place it at dot row 65, which is the same as graphics row 9, and dot column 41 which is where all the bars start. Now we will add 1 to  $C$  so that if we have another value that is less than one full block we will redefine character 98. If your bar graph values are all less than one full block this program will redefine character numbers 97 thru 117. To prevent this I recommend that you multiply all your values by 10 before entering them into  $G(I)$  and then indicate in your graph title that you have done this.

---

270 Here is where we increment I until we have finished displaying our bar titles and bars.: In our example program when I=19 we will leave the loop and the subprogram and jump back to line 150.

## GENERAL BAR GRAPHER SECTION 2

Now that you have seen the example program run you might want to use your own bar titles and values. This can be done by changing the DATA statements or you can replace lines 110, 120 and 130 with the following code.

```
110 ON WARNING NEXT :: DISPL
AY AT(1,1):"G$(0)=" :: ACCEP
T AT(1,8):G$(0)
```

```
120 FOR I=1 TO 20 :: DISPLAY
AT(I+1,1):"G$(" & STR$(I) & ")="
" :: ACCEPT AT(I+1,8)SIZE(7)
:G$(I):: IF G$(I)=" THEN 14
0
```

```
130 DISPLAY AT(I+1,16):"VALU
E=" :: ACCEPT AT(I+1,22):G(I
):: NEXT I
```

When you run the program with these new lines of code it will display input prompts on the screen. The first prompt is G\$(0) and is used for the bar graph title. After the bar graph title has been input, it will generate input prompts for G\$(1) thru G\$(20) and G(1) thru G(20). The G\$'s are used for the bar titles and the G variables are for the value to be displayed by the bar grapher. You may input less than 20 items by pressing enter when the G\$ prompt appears. This will create a null or empty string which will cause the program to leave the input mode and jump down to execute the subprogram GRAPH.

## ADDENDUM

Listed below are the program lines that may be changed to modify the General Bar Graph program to show a scale that is lower than 0 thru 200. Changing and adding the following lines will allow a low scale of 0 thru .25. The balance of the program listing is on pages 63 thru 65.

```
160 SUB GRAPH(G$(),G()):: CA
LL CLEAR :: CALL SCREEN(7)::
FOR I=1 TO 8 :: CALL COLOR(
```



---

```
I,2,12):: NEXT I :: CALL COL  
OR(12,13,4,9,13,4)
```

```
170 CALL CHAR(124,"010101010  
10101010000000001",62,"00000  
0000001",94,"00000001010101"  
,96,"00FFFFFFFFFFFFFFF")
```

```
180 M=0 :: C$=RPT$(RPT$(CHR$(  
(125),4)&CHR$(124),5):: DISP  
LAY AT(1,(29-LEN(G$(0)))/2):  
G$(0): : : " ^>>>>^>>>>^>>>>  
^>>>>^>>>>^"
```

```
190 FOR I=1 TO 20 :: IF G$(I  
<>" " THEN DISPLAY AT(I+4,4)  
:C$ :: M=MAX(M,G(I))ELSE 205
```

```
205 IF M>8 THEN C=1 :: GOTO  
210 ELSE C=100 :: M=M*100
```

```
206 FOR I=1 TO I-1 :: G(I)=G  
(I)*100 :: NEXT I
```

```
215 IF M>100 THEN M=INT((M-1  
) /200)+1 ELSE IF M>50 THEN M  
=.50 ELSE IF M>25 THEN M=.25  
ELSE M=.125
```

```
220 DISPLAY AT(2,1):"SCALE X  
"&STR$(MX)&" > ="&STR$(M*8/C  
)
```

```
225 IF C=100 THEN DISPLAY AT  
(3,3):USING "0 #.## #.## #.  
# #.## #.##":40/MX*M/C,80/MX  
*M/C,120/MX*M/C,160/MX*M/C,2  
00/MX*M/C :: GOTO 240
```

```
240 C=0 :: FOR I=1 TO I-1 ::  
DISPLAY AT(I+4,1)SIZE(3):G$(  
I):: IF G(I)<M*8 THEN 260
```

# **FOR YOUR TI 99/4 AND TI 99/4A COMPUTER**

## **Minimum System Requirements**

- TI 99/4 or 99/4A Console
- Monitor or T.V.
- Cassette Player
- Extended Basic Command Module

This guide will show you some of our professional programming secrets on how to:

- Use CALL PEEK
- Get Sprites to pick up objects, eat dots and lay down a trail.
- Shoot sprites without missing a coincidence.
- Make one sprite chase another.
- Generate moving sprite patterns.
- Easily convert sprite rows and columns into graphic rows and columns and visa versa.
- Use 3 different CALL KEY or CALL JOYST examples for moving sprites.
- Write a GENERAL BAR GRAPHING program (to one pixel accuracy) that shows you sprites aren't just for games.

Full of fast running and Byte saving examples that you can use in your existing programs or combine together to write your own programs. Each example program is fully documented in a step by step method that is easy to understand.