Vol. 1 / Number 2 April 1985

The Adam Technical Journal

In this issue:

Introduction to graphics sprites.................... 15

Defining sprite shapes and colors 17

Magnification of sprite shapes 19

Controlling joysticks programmatically 19

Additions to the Basic Utility Library 1.0.......... 20

Example programs to demonstrate new utilities 22

Modify SMARTBASIC high resolution graphics mode 23

More helpful Tid-Bits about the ADAM 25

Copyright 1985, Serendipity Productions

The ADAM TECHNICAL JOURNAL is published bi-monthly by Serendipity Productions. Subscription rates are $18.00 per year in the U.S. and Canada, $23.00 per year in any other country, payable by check or money order only. Single issues are available for $5.00. All inquiries and payments should be made to Serendipity Productions, P.O. Box 07592, Milwaukee, WI 53207.

Preview of the next issue:

- Introduction to the SMARTBASIC interpreter
- Examine the CPU representation of a Basic program
- We'll explore the four main reserved word tables
- See how the interpreter executes a program
- All concepts are demonstrated in example programs
- We'll add a new command to SMARTBASIC to make our utilities easier to use
- More useful Tid-Bits of information

BI MONTHLY NEWSLETTER

VOL 1 -NO. 2 4/85

CREATED FOR COLECO ADAM OWNERS

A TECHNICAL NEWSLETTER ON HOW TO
OUT SMART "SMART BASIC"

SERENDIPITY PRODUCTIONS
P.O. BOX 07592
MILWAUKEE, WI 53207

INTRODUCTION

We assume you have read and understand the material presented in Vol 1/1. If you aren't comfortable with the way the VDP operates or how our utilities work with it, this issue will be rather hard to follow. If this is the first issue of The Adam Technical Journal that you've received, we strongly recommend that you get a copy of Vol 1/1 since it lays quite a bit of the ground work for the topics we are going to cover here.

We also recommend for those who are serious about using all the capabilities of the VDP, the TMS 9918A manual available from Texas Instruments by calling the Customer Response Center at 1-800-232-3200, or by writing to:

Texas Instruments
P.O. Box 225012, M.S. 57
Dallas, Texas 75265.

This issue will continue our discussion of the TMS 9918A Video Display Processor (VDP) used by the Adam computer for all the graphics display on your monitor.

In the first issue of the Journal we covered how to redefine characters, display the characters on the screen and how to manipulate the screen and character colors. The altering of these tables was done with respect to the normal graphics mode of SmartBasic. This mode divides the screen into 24 rows by 32 columns.

In this issue we will cover objects called sprites in relation to both normal Smart Basic text mode and the high resolution graphics mode. We will give you the ability to plot in HGR using any of the 16 colors that the VDP allows. The joystick will also be briefly discussed. Lastly, new utilities and example programs will be provided.

SPRITES

Sprites are animated patterns that provide smooth motion and allow multilevel pattern overlays. These objects are defined much the same way a character is defined. In fact, we will use the character definition subroutine to define them. The VDP will support up to 32 sprites. These are used in most of the arcade video games that are produced for the Adam.

Sprites can be moved one pixel at a time on the screen. NOTE: A pixel is the smallest position on the screen and when turned on appears as a dot. When the sprites are moved, the screen which was produced in text graphics or HGR mode will not be disturbed. The 32 sprites also can overlay each other giving the illusion of a three dimensional free moving object. Sprites are numbered 0-31 with 0 being the highest priority sprite. This means if you overlay sprite 0 on sprite one it will appear that sprite 0 is in front of sprite one. This pattern will continue for higher numbered sprites.

Sprites can have any one of the sixteen colors the VDP allows. Only the foreground color is definable. The background color is always set to transparent.

Sprites can be magnified 1, 2, and 4 times their normal size. This can be used to make the sprite appear to move away or towards you.

VDP REGISTERS

VDP registers are a type of fast memory location that are incorporated into the Video Display Processor. These registers hold 14 bits of data plus two bits which describe what is to be done with the data (ie, write to VREG, write to VRAM or read from VRAM). These two control bits are the first two in the first byte. They make up the offset which we discussed in the last issue (page 2, column 2). For example, in the first issue of this journal we had to change the contents of register 7 in order to modify the screen color. In this issue we will need to alter registers 1, 5, and 6. For our purpose, these registers control: sprite character size (register 1), base address of the sprite attribute table (register 5), and the base address of the sprite pattern generator table (register 6).

The VDP registers are changed through port 191, this being the same port used for setting the VDP memory write and read addresses discussed in the last issue. In fact we will be using the same assembly language routine and many of the basic utilities from the last issue. We will be adding the sprite utilities to these utilities later.

All that is necessary to change VDP registers is to obtain a decimal value as an address to be used in the POKEVM subroutine. This value contains information as to the register to be changed and the quantity to be placed into the register.

This decimal value can be computed by looking at the bits of the two bytes that are transferred through port 191 to change a register. The high byte always contains 10000 as the first five bits; the first two indicating a VREG write and the next three being zero since they are not used for this register. The last three bits are used to indicate which of the eight registers are to be changed. The low byte is used for transmission of the value you wish to put into the eight bit register.

```
|1|0|0|0|0|R|E|G| |D|A|T|A| | | | |
```

To illustrate the creation of a decimal value to change a register we will look at the subroutine to change screen color from the last issue. First of all we need to determine a number to represent register 7 with zero being its contents. This is simply done by setting the appropriate bits. The first and second bytes are diagrammed below.

```
|1|0|0|0|0|1|1|1| |0|0|0|0|0|0|0|0|
{   128   }{  7  } {       0       }
      135
```

The high byte is evaluated to be 135 and the low byte is 0. Converting this into a single number we obtain 135* 256+0=34560. You can change the screen color by taking this number and adding any one of the 16 color codes to it and then transmitting the low and high bytes through port 191.

Now let's look at line 57000 of the last issue's utilities. You will see that we are not using 34560 but 18176 as the base number. We are doing this because this subroutine calls the POKEVM subroutine found at line 52000. In this routine a 64 is added to the address to indicate a VDP memory write. Therefore to compensate for the 64 offset we must subtract it from the high byte before multiplying by 256. The equation now reads (135-64)*256+0= 18176. You will also see in line 57000 two pokes before the GOSUB to 52000 and two more pokes after. The first two pokes disable the writing of any data to VDP memory that would be caused by POKEVM assembly language during the changing of a register. The last pokes reenable this function. We will be using

subroutine 57000 for changing registers in the future. Therefore, all register calculations will take into account an offset of 18176.

There is one more piece of information we should clarify before we continue. This is the instruction POKE 17009,0 that you will find in any subroutine that accesses VDP memory. This location is memory mapped to the VDP and when set to zero tells the VDP that there will be information coming through ports 190 and 191. Without clearing this location there will be occasions when data will appear at an address in VDP memory other than the one you indicated.

SETTING UP SPRITE TABLES

As we mentioned earlier there are two tables that are required to define sprites. The first table is the Sprite Pattern Table. This table is used for sprites in the same way as the character definition table that was discussed in the last issue. The VDP allows us to determine where in VDP memory we would like this 2048 byte table to be located by the use of VDP register 6. For the sprite utilities we have decided to place this table starting at VDP memory location 14336. You can see there is no possible way we can fit 14336 into a one byte register that can have a maximum size of 255. The VDP only needs three bits to determine this location. The reason is they have, for this register, divided the 16K of memory into eight 2K blocks. By the use of three bits we can choose any one of the 8. Therefore the allowable addresses for the definition tables are:

Reg 6	Location
00	0000
01	2048
02	4096
03	6144
04	8192
05	10240
06	12288
07	14336

We chose 14336 as the starting address because it was out of the range of any other tables used in the high resolution graphics mode and text graphics mode.

Here comes the hard part. We are going to use subroutine 57000 which is the screen color subroutine to change the contents of register 6. Not only do we have to worry about the 64 offset due to the POKEVM routine, but also the offset caused by register 7 being the base register. First we need to calculate the two bytes needed for changing register 6 to point to VDP memory 14336. We must place the number seven in register six to indicate this block of memory. Therefore, the bit pattern for the high and low bytes are:

```
|1|0|0|0|0|1|1|0|  |0|0|0|0|0|1|1|1|
       reg 6           7th block
```

The high byte converts to 134 and the low byte to 7. If we subtract the 64 offset from the high byte and convert the two bytes to a single decimal value we arrive at the value (134-64)* 256+7=17927. To use subroutine 57000 we must subtract the offset 18176 due to register 7. The new value becomes 17927-18176= -249. You will find -249 is one of the numbers being loaded into variable (uc) in subroutine 59500 which is the subroutine that sets all registers concerning sprites.

Although we chose 14336 as this table location you could choose any one of the other 7 locations. There is an advantage to choosing location 0 when working in the graphics text mode. As you remember from last time, the first 2K of VDP memory is used as the character definition table by the graphics text mode. If we also use this table for sprite patterns we will automatically have all the sprites defined as the ASCII character set. This will be illustrated in an example program.

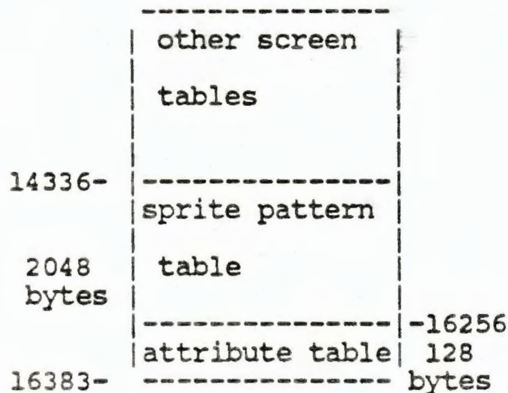
The second table we must define is the sprite attribute table. This table contains 32 entries of 4 bytes each, a total of 128 bytes. Just like the sprite pattern table, this table's location is defined by a VDP register. If a value between 0 and

127 is put into register 5, one of the 128, 128-byte blocks of VDP memory will be allocated for the sprite attribute table. We have decided to put this table in the last 128 bytes of the sprite pattern table. This corresponds to VDP memory location 16256 and a value of 127 to be put in register 5. It should be noted that tables can be overlapped to conserve VRAM space. The sprite pattern table is 2048 bytes in length but if you never intend to use all 2048 there's no reason you can't put any other table on top of the unused portion. Its up to the programmer to manage the use of these tables so they don't conflict. Below you'll find the calculations needed to use subroutine 57000 for register 5.

```
|1|C|0|0|0|1|0|1|  |0|1|1|1|1|1|1|1|
        reg 5             127
     133
```

uc=((133-64)* 256+127)-18176)=-385

The following figure illustrates how these tables are located in VDP memory.



SPRITE PATTERN TABLE ENTRIES

So what good is what we have just discussed in the last section? Well, first of all, it's an explanation of how to define the VDP register. Secondly, it leads us into what type of information must be put into these tables to get sprites on the screen.

The pattern table is used to define the appearance of the sprite. The sprite definition uses the short hand form described in VOL 1/1 of this journal. In fact, sprite definition is identical to character definition with the exception that the two tables are defined in different locations in memory.

SPRITE ATTRIBUTE TABLE ENTRIES

Any one of the 32 4-byte entries in this table gives information to the VDP on where to place the sprite, which pattern definition entry is to be used, and the color you wish the sprite to be. Below you will see a figure illustrating an entry in this table.

byte 0	Vertical Position
byte 1	Horizontal Position
byte 2	Pattern Name
byte 3	Color

When the sprite is painted on the screen, the upper lefthand corner is the reference point for the VDP.

The screen is divided into 256 pixels in the horizontal direction and 192 pixels in the vertical direction. If you were to put a 96 in the byte that defines the vertical position and a 127 in the byte that defines the horizontal position, and if you were to set the proper pattern name and color code, a sprite will appear in approximately the middle of the screen. So let's get a sprite on the screen. Note that this is just an example of the mechanics used in the upcoming utilities.

We first need to load the utilities from the last issue. We need the POKEVM and Screen Color utilities to set up the sprites and their tables. The short program that follows sets the sprite pattern table to start at VDP memory location zero and the sprite attribute table to 16256. Since the sprite definitions are the same format as the character definitions, setting the pattern table location to 0 in effect defines sprite pattern to be the

ASCII set. We will define sprite zero to be located in the middle of the screen as a red letter "A". NOTE: SEE UPDATE SECTION FOR CHANGES TO UTILITIES.

EXAMPLE #1:

```
 90 HOME : REM CLEAR SCREEN
100 GOSUB 51000 :REM SET UP
ASSEMBLY LANGUAGE
200 us=-385 : GOSUB 57000 : REM SET
REG 5 TO LOCATE SPRITE ATTRIBUTE
TABLE AT MEMORY LOCATION 16256
300 us=-256 :GOSUB 57000 : REM SET
REG 6 TO LOCATE SPRITE PATTERN
TABLE AT MEMORY LOCATION ZERO
400 ua=16256 : ub=96 : uc=1 : GOSUB
52000 :REM POKEVM VERTICAL POSITION
OF SPRITE ZERO
500 ua=16257 : ub=127 : uc=1 :
GOSUB 52000 : REM POKEVM HORIZONTAL
POSITION OF SPRITE ZERO
600 ua=16258 : ub=65 : uc=1 : GOSUB
52000 : REM POKEVM PATTERN NUMBER.
IN THIS CASE THE ASCII CODE FOR THE
LETTER "A"
700 ua=16259 : ub=6 : uc=1 : GOSUB
52000 : REM POKEVM COLOR OF SPRITE
ZERO. IN THIS CASE RED. SEE VOL 1/1
FOR COLOR CODE NUMBERS.
800 END
```

You will see a red letter appear in the middle of the screen after execution of this program. If you wish to define other sprites all that is needed is to define the next 4 consecutive bytes as we did with sprite zero.

We mentioned in the last journal that you can reinitialize VDP memory by executing the TEXT command. Well, this was true for the tables we covered back then. Now however we are creating sprite tables beyond what can be reinitialized this way. Therefore, the red sprite from the example program will be there until you either redefine it or you shutdown the computer. To clear the screen of sprites you could use the POKEVM routine to zero out the entire attribute table. This line would read: ua=16256 : ub=0 : uc=127 : GOSUB 52000

SPRITE MAGNIFICATION

You don't have to be satisfied with a sprite covering only an 8 x 8 pixels square. The TI chip allows you to double and even quadrupal the size of a sprite. This is done simply by placing a value between zero and three in the two least significant bits of register one of the VDP. A zero will cause all sprites to be the normal 8 x 8 pixel size. A one will double the size of the 8 x 8 pixel sprite by making each pixel appear as a 4 x 4 pixel square. A two will cause the pattern table to be split into 32 blocks of 4 patterns. These four individual patterns can be defined to form a single 16 x 16 pixel sprite. Lastly, a three causes the sprite which has it's pattern defined with a four pattern entry like that of option 2 will be doubled in size such as used in option 1. What this does is give you a sprite the size of 32 x 32 pixels.

This might be a good time to mention two problems with sprites. First, no more than four sprites can appear on the same horizontal plane. Anymore thereafter will not be seen. The second problem is that you cannot control the magnification of an individual sprite without altering the size of the rest.

The register evaluation for register 1 is done in the same way we did register 5, and 6. The number we arrive at is -1340. Adding 0-3 to this value and placing it into variable (us) and GOSUB 57000 will change the magnification of all sprites on the screen.

MEMORY LOCATIONS FOR JOYSTICKS

The joysticks are memory mapped to several locations. We will in this newsletter be concerned with the PDL numbers 0,1,2, and 3. The other PDLs control the key pads and fire buttons which can be easily accessed by the PDL commands of Smart Basic.

The difference between PDL 0,1,2,3 and the others are that these are actually counters that can be used to return the vertical and horizontal positions of the joysticks. The value of the counters are changed when (and only

when) a PDL command is issued AND the joystick is not centered on the hand-set. The numbers that are generated in the horizontal and vertical are from 0 to 255. Although this makes it easy to use for moving sprites around the screen by simply poking the values of the joystick into the attribute table, it does create two problems.

The first is that the vertical position of a sprite should not exceed 192. If it does, the sprite will disappear off the bottom of the screen and there will be a delay before you can make it reappear again. The delay is due to the time required for the joystick counter to count backwards from 255 to 192.

The second problem is that the joystick counter will change any time a PDL command is encountered. So if you check for a firebutton or a keypad result, all the paddle counters will increment if the joystick is not centered. This caused a problem in the example PAINT program where points were skipped while trying to draw a continuous line.

To solve these problems we need to be able to change the counters in some way other than by the use of the joystick. This is done by poking any value you wish between 0 and 255 right into the four memory locations mapped for these PDLs. These locations are in CPU memory at:

PDL(0)	27100
PDL(1)	27102
PDL(2)	27101
PDL(3)	27103

An example is if the PDL(0) counter is presently at 192 and we would like to restart at zero. All that is required is to POKE 27100,0. Further examples will be given later.

SPRITE UTILITIES

Here is the fun stuff. We will be defining 6 new utilities. The format of what is coming is first a brief description of what the utility does. Second will be the format of the GOSUB. Lastly, the new basic utility code will be given to add to the utilities from the last time.

SPRITE MAGNIFY AND SETUP
GOSUB 59500

This subroutine sets up the registers that contol the sprite pattern and attribute tables. It also changes the magnification factor of sprites being displayed, given a number 0 to 3 in variable (um). NOTE: SEE THE UPDATE SECTION AT THE END OF THIS ISSUE.

```
FORMAT  um=3 : GOSUB 59500
To just magnify you may also do:
        um=3 : GOSUB 59520
```

UTILITY (1)

```
      SET UP SPRITE PATTERN
      TABLE R6
59500 us=-385 : GOSUB 57000

      SET UP SPRITE ATTRIBUTE
      TABLE R5
59510 us=-249 : GOSUB 57000

      SET R1 FOR SPRITE
      MAGNIFICATION
59520 us=-1340+um : GOSUB 57000
59530 RETURN
```

SPRITE PATTERN DEFINITION
GOSUB 59800

This subroutine is the same as the character definition subroutine from the last time with the exception of putting a 1792 character offset on the variable (us). The offset pushes (us) into the sprite pattern table.

If you assign (us) the pattern number (similar to an ASCII CODE) and (us$) equals the shorthand code of the pattern, and do a GOSUB 59800, a pattern will be entered in the pattern table to be used by the next utility we will talk about.

```
FORMAT  us=65 :
        us$="123456789ABCDEF"
        GOSUB 59800
```

UTILITY (2)

```
59800 us=us+1792 : GOSUB 54000 :
      RETURN
```

SPRITE DEFINITION
GOSUB 59000

This utility assigns a pattern from the pattern table to a sprite number. By setting (us) to the sprite number and (ua) to the pattern number just defined in the last utility, and performing a GOSUB 59000, you will define the appearance of a sprite.

FORMAT us=0 : ua=65 : GOSUB 59000

UTILITY (3)

```
59000 POKE ux+10,127 : POKE ux+6,us
      *4+130 : POKE ux+20,ua : POKE
      ux+3,1 : POKE ux+4,0 : POKE
      17009,0 : CALL ux : RETURN
```

Although this looks quite complicated it is really doing nothing more than what the POKEVM routine does when it calls the assembly language program. The reason this line is doing all the work rather than calling GOSUB 52000 is to save execution time. The next several utilities will also access the assembly program directly.

SPRITE LOCATE
GOSUB 59600

This utility allows you to set the position of a sprite on the screen. Variable (us) equals sprite number and variables (ur) and (uc) are used to hold the row and column positions of the upper lefthand corner of a sprite. (ur) and (uc) can have a range between 0 and 255.

FORMAT ur=96 : uc= 128 : us=0 :
GOSUB 59600

UTILITY (4)

```
      PUT ROW VALUE IN ATTRIBUTE
      TABLE
59600 POKE ux+10,127 : POKE ux+6,us
      *4+128 : POKE ux+20,ur : POKE
      ux+3,1 : POKE ux+4,0 : POKE
      17009,0 : CALL ux
      PUT COLUMN VALUE IN ATTRIBUTE
      TABLE
59640 POKE 17009,0 : POKE ux+20,uc
      : POKE ux+6,us*4+129 : CALL
      ux
59650 RETURN
```

Again, the complexity of this utility is justified to increase the speed of execution.

SPRITE POSITION
GOSUB 59700

This utility returns the position of a sprite, given the sprite number is in variable (us) and a GOSUB 59700 is performed. The row location is returned in variable (ur) and the column location is returned in variable (uc).

FORMAT us=0 : GOSUB 59700 : row=ur
: col=uc

UTILITY (5)

```
      PEEKV THE ROW LOCATION
59700 POKE 17009,0 : POKE ux+33,us*
      4+128 : POKE ux+37,63 : CALL
      ux+31 : ur=PEEK(ux+55)

      PEEKV THE COLUMN LOCATION
59750 POKE 17009,0 : POKE ux+33,us*
      4+129 : CALL ux+31 : uc=PEEK(
      ux+55)
59760 RETURN
```

SPRITE COLOR
GOSUB 58500

This utility allows you to change the color of a sprite. Variable (us) is used for the sprite number and variable (uc) is used to hold the sprite's color code. By doing a GOSUB 58500 you will change the color of the sprite.

FORMAT us=0 : uc=5 : GOSUB 58500

UTILITY (6)

```
58500 POKE xu+10,127 : POKE ux+6,us
      *4+131 : POKE ux+20,uc : POKE
      ux+3,1 : POKE ux+4,0 : POKE
      17009,0 : CALL ux : RETURN
```

EXAMPLE PROGRAMS

This program demonstrates how to define a sprite pattern and move sprites on the screen

```
 90 REM SET UP UTILITIES
100 GOSUB 51000
110 HOME : CLEAR SCREEN
115 REM DEFINE CIRCLE SPRITE
    PATTERN
120 us=0 : us$="001C3E7F7F7F3E1C" :
    GOSUB 59800
130 REM SET UP SPRITE TABLES
140 um=0 : GOSUB 59500
144 REM SET COLOR TO BLUE
145 us=0 : uc=5 : GOSUB 58500
150 REM ASSIGN PATTERN 0 TO SPRITE
155 us=0 : ua=0 : GOSUB 59000
156 REM MOVE SPRITE DOWN SCREEN
160 FOR ur=0 TO 192 : uc=126 :
    GOSUB 59600
170 NEXT ur
180 GOTO 160
```

PUT UTILITIES HERE

If you would like the sprite to be twice the size, change the (um) in line 140 to equal 1.

This example program demonstrates the use of the magnify subroutine. Two patterns of a little man are created. The smaller sprite used one pattern number and the larger man requires four pattern numbers. These patterns are exchanged and magnified to make the little man appear to move forwards and backwards.

```
 90 HOME : REM CLEAR SCREEN
100 GOSUB 51000 : REM SET UP
    UTILITIES
110 REM DEFINE 8X8 PIXEL MAN
120 us=0 : us$="1038107C92284444" :
    GOSUB 59800
130 REM DEFINE 16X16 PIXEL MAN
140 us=4 : us$="01030301070F1D31" :
    GOSUB 59800
150 us=5 : us$="61010103060C1818" :
    GOSUB 59800
160 us=6 : us$="80C0C080E0F0D8CC" :
    GOSUB 59800
170 us=7 : us$="C6C0C06030180C0C" :
    GOSUB 59800
180 REM LOCATE SPRITE IN MIDDLE
    OF SCREEN
190 us=0 : ur=96 : uc=126 : GOSUB
    59600
195 REM ASSIGN SPRITE 0
197 us=0 : uc=6 : GOSUB 58500 : REM
    SPRITE COLOR
200 us=0 : ua=0 : GOSUB 59000
210 um=0 : GOSUB 59500
225 FOR i=1 to 100 : NEXT i : REM
    DELAY
226 us=0 : ua=8 : GOSUB 59000 : REM
    CLEAR SPRITE
228 REM ALTERNATE MAGNIFICATION
230 um=2 : GOSUB 59500
232 us=0 : ua=4 : GOSUB 59000
235 FOR i=1 TO 100 : NEXT i
240 um=3 : GOSUB 59500
250 FOR i=1 TO 400 : NEXT i
260 GOTO 200
```

PUT UTILITIES HERE

This example defines 4 sprites with patterns from the character definition table. The sprites are lined up on either side of the screen. The zero sprite is located in the middle of the screen and is controlled by joystick (1). The objective is to keep away from all the other sprites. This is made difficult because the other 4 sprites are tracking the one you are controlling through the use of the sprite position utility. The four sprites will try to converge on you. The program ends when any sprite is within 8 pixels of you.

```
 90 HOME : REM CLEAR SCREEN
100 GOSUB 51000
110 um=1 : GOSUB 59500 : REM
    MAGNIFY SPRITES
115 REM TRANSFER CHARACTER
    DEFINITION TABLE TO SPRITE
    PATTERN TABLE
120 FOR i=512 TO 768+5*8
130 ua=i : GOSUB 56000 : REM PEEKV
140 ua=i+14336 : uc=1 : GOSUB 52000
    : REM POKEVM
150 NEXT i
155 REM ASSIGN SPRITES
160 FOR i=0 TO 4
170 us=i : ua=i+64 : GOSUB 59000
175 us=i : uc=5 : GOSUB 58500 : REM
    SPRITE COLOR
180 NEXT i
185 REM LOCATE SPRITES
190 us=1 : uc=10 : ur=10 : GOSUB
    59600
200 us=2 : uc=10 : ur=180 : GOSUB
    59600
```

```
210 us=3 : uc=210 : ur=10 : GOSUB
    59600
220 us=4 : uc=210 : ur=180 : GOSUB
    59600
280 us=0 : uc=7 : GOSUB 58500 : REM
    SET COLOR
290 POKE 27102,96 : POKE 27103,127
    : REM INITIALIZE JOYSTICK (1)
295 REM CONTROL CENTER SPRITE
300 us=0 : ur=PDL(1) : uc=PDL(3) :
    GOSUB 59600
310 r=PDL(1) : REM CAUSE SPRITE TO
    MOVE MORE THAN ONE PIXEL
315 c=PDL(3)
319 REM MOVE 4 SPRITES TO CONVERGE
    ON CENTER SPRITE
320 FOR i=1 TO 4
330 us=i : GOSUB 59700 : REM GET
    POSITION OF SRITE
340 t=((ur-r)^2+(uc-c)^2)^.5 : REM
    CALCULATE DISTANCE FROM CENTER
    SPRITE
350 IF t<8 THEN 500 : REM END
    PROGRAM IF SPRITE IS WITHIN 8
    PIXELS
360 t=r-ur : REM CALCULATE
    DIRECTION TO MOVE
370 IF t>0 THEN ur=ur+4 : GOTO 390
380 ur=ur-4
390 t=c-uc
400 IF t>0 THEN uc=uc+4 : GOTO 420
410 uc=uc-4
420 us=i : GOSUB 59600 : REM LOCATE
    SPRITE
430 NEXT i
440 GOTO 300
500 PRINT "YOU ARE HIT"
510 END
```

The last example program, called PAINT, illustrates sprites used in the HGR mode. We will demonstrate most of the sprite utilities and how to obtain all 16 colors while plotting. The program listing is at the back of this issue because of its length. The utilities are not included to save space so don't forget to add them . A description of the program and its use can be found below. But before we do that, no discussion of graphics would be complete without covering SmartBasic's Hi-Res graphics capabilities (HGR).

SMARTBASIC HI-RES

As we mentioned, the ADAM does not allow all sixteen colors in the HCOLOR command. The reason is that in Apple Soft Basic only 4 colors can be specified. The authors of Smart Basic must have felt that maintaining compatibility with Applesoft Basic was more important than giving you all of the VDP capabilities. Fixing this problem is simple. All we need do is poke the color code number discussed in the last issue into CPU memory location 16777 instead of using the HCOLOR command. This is done on line 2000 of the example program. Whatever color code is placed at 16777 will show up when you use the HPLOT command.

Another problem with HGR is that the Screen Color utility will not work. The reason is that the HGR command sets up the color table so that the background colors and screen are black. To solve this problem all we need to do is to poke into CPU memory the color code for transparent at locations 25431 and 25471 before performing HGR. You can see we did this at line 10 of the example program.

Two other problems with HGR will become apparent when using our PAINT example. One is that ADAM has not given us the ability to erase a line. Also, when two color lines get too close together there is "bleeding" between them. In order to understand why, we must examine how the VDP performs high resolution graphics.

The main difference between displaying text and graphics is the size of the tables in VRAM. To display text, we define the shape of each character in a pattern table and the color of groups of 8 characters in a color table. There is a name table which holds one byte for each of the 768 text positions on the screen (remember last issue?). We simply enter the character name (position in the pattern table) in the appropriate position in the name table and that character appears on the screen. This is really like a low resolution graphics since the smallest level of detail one can specify is one character (8x8 pixels); unless we change the character patterns.

In Hi-Res, things work the same except now we really are changing the patterns. Instead of defining a pattern for each of the 90 or so printable letters, we define a 1-byte pattern for every 8x1 pixel area of the screen, 1 bit for every pixel. Instead of a small color table with one byte for 8 characters, we use a large table with 1 byte for every 8x1 pattern. The first 4 bits indicate the color of the pixel when turned off (background color) and the last 4 indicate the color when on (foreground). Since there is a unique pattern for every 8x1 area of the screen, we never change the name table.

When the HPLOT command is used to draw a line, Smart Basic calculates which patterns have to be changed, then changes only those bits (pixels) necessary. Now come the flaws we alluded to above. Smart Basic initializes the tables when the HGR command is used. It sets the background color to black instead of transparent, hence the pokes to 25431 and 25471 noted above are needed so the screen color will be visible behind the character patterns. The next flaw is the inability to erase a line once drawn. The best you can do is PLOT over it in the same color as the background, but that isn't a true erase because you haven't turned the pixel off. You've only changed its color to match the surroundings. You're still vulnerable to color "bleeding" which is the next item.

In Hi-Res, each entry in the color table defines the background and foreground colors for a cell of 8x1 pixels. As far as color is concerned, that's the resolution limit. Let's say we draw a blue horizontal line in the middle of the screen, followed by a red vertical line so that they cross in the center. Smart Basic sets the color definitions for each cell containing a pixel for the red line. When it gets to the intersection, it will set the foreground color for that 8x1 block of pixels to red. The trouble is that the other 7 of the eight bits are also turned on and are supposed to be blue, but are now red. This is color "bleed". If we had "erased" the blue line as discussed above before drawing the red one, the problem would still exist because the pixels are still on and are supposed to be black (or transparent).

HOW TO USE THE PAINT PROGRAM

This program gives you the ability to draw on the screen using different colors and brush widths. Along with this you can plot circles, triangles, squares, connect up to nine points, change screen color, and clear the screen to start over.

After entering RUN there will be a delay for sprite setup before you will be given control through joystick (1). The right fire button creates and removes the palette. When the palette is displayed you will be given a pointer sprite that is controlled by the joystick. To choose a color or option place the pointer next to it and press the left fire button.

The horizontal and vertical line option allows you to set the length and width of the line you are going to draw. When you choose either line option and hit the left fire button the program will wait until you choose a number on the numeric pad. If you choose the width and length to be numbers greater than 2, the lines that are painted will appear to be three dimensional.

The triangle option will connect the last three points plotted. The square option will connect into a square the last two diagonal points plotted.

The open circle option will use the last two points plotted as the center and radius of a circle. The circle will be plotted on the screen until it is complete or you press the right fire botton. NOTE: Make certain that center and radius points don't cause the circle to be plotted off the screen. The closed circle option is the same as the open with the exception that a solid circle is plotted.

The "S" option changes the screen to whatever color is presently on the brush.

The "C/L" option clears the screen. After execution there will be a slight delay and an over sized palette will be displayed. To initialize the palette, press the right fire button.

The "C/P" option will connect up to 9 points. While you are plotting points the program is keeping track of the last 9 points. When you choose this option the program will wait for you to press one of the number keys on the numeric pad. The number you choose will be the number of points that are connected.

To plot on the screen simply press the right fire button to remove the palette. Make certain that you have chosen a color for your brush. You can move the sprite brush around the screen at a fast rate. When ever you press the left fire button a line or point will be left.

UPDATES AND CORRECTIONS

In our first issue, one of our readers found an inconsistency between the explanation of the screen color utility's description (page 9) and the utility itself. The problem is the description states you must put the color code number in variable (us) before doing a GOSUB 57000. This was incorrect. The variable to be set is supposed to be (uc). This correction was made in the second printing.

We regret that we must confuse the matter more. In order for the sprite table and magnification utility to work we must alter the screen color utility so that (us) is the variable used to pass the color code. Line 57000 must be changed to:

```
57000 ua=18176+us : POKE ux+16,24 :
      POKE ux+17,10 : GOSUB 52000:
      POKE ux+16,227 : POKE ux+17,
      227 : RETURN
```

Several other typos were found and are as follows:

Page 2,column 2, line 14 should read "port #190" not 192. Page 4, column 1, under the heading TABLE(2) for row 1 should read "6144-6175" and for row 3 should read "6208-6239". Page 14, listing #2, byte count 24 the decimal value should be 11 not 12. Also, byte count 29 was incorrectly written as 39. In listing #2, byte count 18, the argument "$0000" should be in parentheses.The comment in the line above should be to GET not SEND data. In both listings, there is an extraneous "F" in the argument of the first OUT instruction.

TID-BITS

Last month we reported that DATA and REM statements grow by one character every time a basic program is read from tape or a line is edited on the screen. One of our readers wrote to tell us that ending a DATA statement with a comma solved the problem. We tried it but the problem is still there. If it does work for other ADAM owners, it suggests there are different versions of SmartBasic in circulation. A PEEK to CPU address 260 will show the version you are presently running. We have version 79 which, to our knowledge, is the most wide-spread.

It is quite common to edit a line in a basic program by listing it on the screen then running the cursor over it, making changes along the way. Here's a few editing aids you most likely know about already but, hopefully, one or two might be new to a few of our readers. Listing a line with a 5 digit line number causes a minor problem because the cursor won't back up over the prompt so it appears that you can't get to column one to copy the entire line. Just press any Smartwriter function, such as DELETE, which will not cause the cursor to advance then backspace which will move the cursor into column 1. Control-N inserts a space in the line at the present cursor location and control-O deletes a character. Control-L clears the entire screen and doing a PRINT CHR$(24) will clear from the current cursor position to the end of the screen.

```
1 REM  ***
3 REM  ***    Program PAINT allows you to
4 REM  ***    paint on the screen using
5 REM  ***    the joystick.
7 REM  ***
10 POKE 25431, 0:POKE 25471, 0:HGR2:REM  Set
HGR background color to transparent
100 GOSUB 51000:REM  Set up utilities
200 um=3:GOSUB 59500:REM  Magnify brush
sprite
240 REM  Set up color and position of palette
245 FOR i=1 TO 15
246 us=i: ua=30:GOSUB 59000
247 uc=32: ur=i*8+8: us=i:GOSUB 59600: NEXT i
248 FOR i=1 TO 9: uc=16: ur=i*8: us=i+15:
GOSUB 59600:NEXT i
250 us=64: us$="C0F07C7E3E3E1F03":GOSUB 59600:
REM  Brush
251 us=65: us$="0100000000000000":GOSUB 59800
252 us=66: us$="0000000000000080":GOSUB 59800
253 us=67: us$="C0E070381C0E0702":GOSUB 59800
254 us=16: ua=7:GOSUB 59000: us=17: ua=9:
GOSUB  59000:REM  Horizontal and vertical line
255 us=7: us$="00FF000000000000":GOSUB 59800 :
REM  Horizontal line
256 us=9: us$="8080808080808080":GOSUB 59800 :
REM  Vertical line
257 us=14: us$="0060908494640407":GOSUB 59800
:REM  Clear screen "C/L"
258 us=15: us$="001C22201C02221C00": GOSUB
59800:REM  "S" screen
260 us=30: us$="FFFFFFFFFFFFFFFF":GOSUB 59800
:REM  Color blocks
270 us=10: us$="007E424242427E00":GOSUB 59800
:REM  Square
275 us=11: us$="0002060A12227E00":GOSUB 59800
:REM  Triangle
278 us=12: us$="001C22414141221C":GOSUB 59800
:REM  Open circle
280 us=13: us$="001C3E7F7F7F3E1C": GOSUB
59800:REM  Closed circle
281 us=17: us$="6090809765070404":GOSUB 59800:
REM  Connect points "C/P"
282 us=18: ua=11:GOSUB 59000:REM Assign
triangle
284 us=19: ua=10:GOSUB 59000:REM Assign
square
286 us=20: ua=12:GOSUB 59000:REM Assign  open
circle
288 us=21: ua=13:GOSUB 59000:REM Assign
closed circle
300 us=0 : ua=64:GOSUB 59000:REM Assign brush
310 us=22: ua=15:GOSUB 59000:REM Assign  "S"
320 us=23: ua=14:GOSUB 59000:REM Assign "C/L"
330 us=24: ua=17:GOSUB 59000:REM Assign "C/P"
400 us=0: uc=2:GOSUB 58500:REM Set color of
brush sprite
450 uc=100: ur=100:REM  Starting location of
brush
451 GOSUB 1900
460 REM  Move brush
499 POKE 27102, ur: ur=PDL(1)
500 POKE 27103, uc: uc=PDL(3)
550 IF ur>180 THEN  ur=180
560 IF uc>245 THEN  uc=245
600 us=0:GOSUB 59600
625 IF PDL(9)=1 THEN GOSUB 1000: GOTO    499:
REM  Check right fire button
699 REM  Check left fire button
700 IF PDL(7)=0 THEN  uc=PDL(3): ur=PDL(1):
GOTO 550
800 HPLOT uc, ur TO uc+er, ur+vr
814 REM  Remember last 9 points
815 IF uc=c(1) AND ur=r(1) THEN 856
825 FOR i=9 TO 1 STEP -1
826 r(i)=r(i-1): c(i)=c(i-1)
827 NEXT i: r(1)=ur: c(1)=uc
856 POKE 27102, ur:POKE 27103, uc
900 GOTO 499
999 REM Set up palette
1000 cu=uc: ru=ur
1002 um=0:GOSUB 59500
1005 FOR us=1 TO 15: uc=us:GOSUB 58500
1020 NEXT us
1030 FOR i=16 TO 24: us=i: uc=3:GOSUB 58500:
NEXT i
1040 REM  Move pointer around palette    and
control joystick
1100 ur=PDL(1): uc=PDL(3): us=0:GOSUB 59600
1101 IF ur<9 THEN POKE 27102, 9
1102 IF uc>40 THEN POKE 27103, 40
1103 IF uc<24 THEN POKE 27103, 24
1104 IF ur>78 THEN POKE 27103, 40
1105 IF ur>134 THEN POKE 27102, 134
1115 IF PDL(9)=1 THEN 1900:REM  Leave
subroutine if right fire button
1120 IF PDL(7)=0 THEN 1100:REM  Continue if
left fire button else loop
1130 co=INT((ur-8)/8):REM  Calculate option
1180 IF uc>30 THEN GOSUB 2000:GOTO 1100: REM
Color
1200 IF co=0 THEN GOSUB 4000:REM Horizontal
brush
1210 IF co=1 THEN GOSUB 3000:REM Vertical
brush
1220 IF co=2 THEN GOSUB 5000:REM Triangle
1230 IF co=3 THEN GOSUB 6000:REM Square
1240 IF co=4 THEN GOSUB 7000:REM  Open circle

1250 IF co=5 THEN GOSUB 7700:REM  Closed
circle
1260 IF co=6 THEN GOSUB 8000:REM  Screen
color
1262 IF co=7 THEN HGR2:REM  Clear screen
1264 IF co=8 THEN GOSUB 9000:REM Connect
points
1270 GOTO 1100
1899 REM  Clear palette
1900 FOR i=1 TO 24
1910 us=i: uc=0:GOSUB 58500
1920 NEXT i
1925 uc=cu: ur=ru
1926 um=3:GOSUB 59500
1930 RETURN
1999 REM  Set brush color. This line allows
all 16 plot colors
2000 POKE 16777, co: we=co:RETURN
2999 REM  Vertical brush
3000 vr=PDL(11)-48
3010 IF vr=-48 THEN 3000
3020 RETURN
3999 REM  Horizontal brush
4000 er=PDL(11)-48
4010 IF er=-48 THEN 4000
4020 RETURN
4999 REM  Use last 3 points to plot triangle
5000 HPLOT c(1), r(1) TO c(2), r(2)
5002 HPLOT c(2), r(2) TO c(3), r(3)
5003 HPLOT c(1), r(1) TO c(3), r(3)
5020 RETURN
5999 REM  Use last 2 points to plot square
6000 HPLOT c(1), r(1) TO c(1), r(2)
6002 HPLOT c(1), r(1) TO c(2), r(1)
6003 HPLOT c(2), r(2) TO c(2), r(1)
6004 HPLOT c(2), r(2) TO c(1), r(2)
6005 RETURN
6999 REM  Use last 2 points produce circle
7000 pi=.0162: w=0:
s=SQR((ABS(r(1)-r(2)))^2+(ABS(c(2)-c(1))^2))
7010 FOR i=1 TO 400:REM  Plot 400 points
7015 IF PDL(9)=1 THEN 7600
7020 w=w+pi
7030 x=SIN(w)*s+c(2)
7035 y=COS(w)*s+r(2)
7036 IF fl=1 THEN 7040
7038 HPLOT x, y:GOTO 7050:REM  Plot open
circle
7040 HPLOT x, y TO c(2), r(2):REM  Plot
closed circle
7050 NEXT i
7600 fl=0:RETURN:REM  Clear closed circle
flag
7700 fl=1:GOTO 7000:REM Closed circle
8000 us=we:GOSUB 57000:RETURN
8999 REM  Plot lines between points
9000 wz=PDL(11)-48
9010 IF wz=-48 THEN 9000:REM  Check numeric
pad
9020 FOR i=1 TO wz-1:REM  Plot points
9030 HPLOT c(i), r(i) TO c(i+1), r(i+1)
9040 NEXT i
9050 HPLOT c(1), r(1) TO c(wz), r(wz)
9060 RETURN
```

Please send my one year subscription of
the **ADAM TECHNICAL JOURNAL** to:

Name: ____

Addr: ____

City: ____________________ State: __________

Zip: __________

1 year (6 issues) - $18 in U.S. and Canada
$23 in other countries

Make check or money order payable to .....

SERENDIPITY PRODUCTIONS
P.O. Box 07592
Milwaukee, WI 53207

Please send my one year subscription of
the **ADAM TECHNICAL JOURNAL** to:

Name: ____

Addr: ____

City: ____________________ State: __________

Zip: __________

1 year (6 issues) - $18 in U.S. and Canada
$23 in other countries

Make check or money order payable to .....

SERENDIPITY PRODUCTIONS
P.O. Box 07592
Milwaukee, WI 53207

SERENDIPITY PRODUCTIONS
P.O. BOX 07592
MILWAUKEE, WI 53207