**COMPUTE! Books** Publication

$12.95

# COMPUTE!'s SECOND BOOK OF ATARI®

From The Editors of **COMPUTE!** Magazine

# COMPUTE!'s SECOND BOOK OF ATARI®

Published by **COMPUTE! Books,**
A Division of Small System Services, Inc.,
Greensboro, North Carolina

A
Small System
Services, Inc.
Publication

Printed in the United States of America

ISBN 0-942386-06-X

10 9 8 7 6 5 4 3 2 1

# INTRODUCTION

Robert Lock, Editor/Publisher, **COMPUTE!** Magazine

Welcome to **COMPUTE!**'s *Second Book of Atari.* This book was a direct result of the overwhelming success of our first book in this series, which is now in its second printing. Unlike our *First Book of Atari*, the *Second Book* is comprised entirely of previously unpublished material. Even if you've followed all of the Atari personal computer information in **COMPUTE!** Magazine since our beginning in the fall of 1979, you'll discover exciting, interesting applications and uses in the pages of this book. And, as always with **COMPUTE!** Publications, you'll find a range of material, from beginner to advanced, ready to type right into your computer – programs and helpful hints designed to teach and entice you, applications and utilities designed to help you better use this fascinating world of personal computing.

We've organized the material and designed the book for ease of use. We welcome your suggestions and comments on this and future titles from **COMPUTE! Books**.

Special thanks to Charles Brannon, Richard Mansfield, and Kathleen Martinek of our editorial staff; Kate Taylor, De Potter, Terry Cash, and Margret Jackson of our typesetting and production staff; Georgia Papadopoulos, Art Director; and Harry Blair, our illustrator.

**COMPUTE! Books** is a division of Small System Services, Inc., publishers of **COMPUTE!** Magazine.
Editorial offices are located at 625 Fulton Street, P.O. Box 5406, Greensboro, NC 27403 USA. (919)275-9809.

# CHAPTER ONE

# UTILITIES

# ATARI BASIC ✓ Joystick Routine

Kirk Gregg

*Complete with many techniques for conserving space and speeding program execution, this handy routine, entirely in BASIC, provides you with a fairly fast, non-assembler dependent, joystick reading routine.*

Two excellent Atari joystick reading routines have been published in **COMPUTE!** (July and August 1981, #14 and #15). So, is there really any need for another Atari joystick routine?

Glad you asked! Both of the published routines work, and both are fast. However, both routines use calls to assembler routines via the USR function call. This method requires that the machine language routine be included in the program encoded as DATA statements to be READ and POKE'd into memory. The code required to initialize the routines and the DATA statements take up space in memory.

There are still many of the original 8K 400s around. If you have one of them, and your program memory requirements begin to approach or exceed your machine's RAM limit, the initialization code and associated DATA statements, used only once in the program, become prime candidates for removal.

Also, since these routines both use calls to USR assembler routines, you have to know Atari/6502 assembler to modify them if necessary for certain applications. I suspect that many of us are on fairly good terms with Atari BASIC, but have not really gotten a handle on assembler just yet.

Therefore, what would be useful is a simple Atari BASIC joystick routine. It should be as compact as possible, for machines with limited memory. Also, it should operate as fast as possible so as not to unduly degrade program speed when used in game applications. So, branches within the routine itself should be avoided, since each transfer of program control to any line number except the next sequential line requires a line number search beginning at the top of program memory, comparing every line number in numerical order until the target line number is located.

Here, then, is my Atari BASIC joystick routine:

```
20  S=STICK(S)
30  DX=(S=5 OR S=6 OR S=7)-(S=9 OR S=10 OR S=
    11)
40  DY=(S=5 OR S=9 OR S=13)-(S=6 OR S=10 OR S
    =14)
50  RETURN
```

x

This routine, as written in the listing above, uses less than 200 bytes of RAM, including variables. It can be combined into one program line, requiring only 188 bytes.

To use this routine, set variable S equal to the number of the joystick you want to read (0-3). Then, call the subroutine, in this case GOSUB 20. The delta-X and delta-Y values are returned in variables DX and DY, respectively. Note that the value input to the routine in variable S is lost. If the input value needs to be retained for use by the calling routine, change the input joystick index parameter variable to any other available variable. For example, if your calling routine needs to read the joystick for each of the four players in a game control loop, it could contain the following sequence:

```
FOR I=0 TO 3
GOSUB 20
```

(Code to handle DX and DY for player 1)

```
NEXT I
```

Then, the joystick routine entry point, line 20, would be changed to:

```
S=STICK(I)
```

In some applications, you may want to get joystick readings for only the four cardinal directions, that is, N-S-E-W, but ignore the diagonal stick readings. In that case, modify lines 30 and 40 of the routine as follows:

```
30  DX=(S=7)-(S=11)
40  DY=(S=13)-(S=14)
```

To try out this routine, type in the routine and the following short demonstration program:

```
10 GOTO 100
100 S=0:GOSUB 20
110 IF NOT (DX OR DY) THEN 100
120 ? ,"dX = ";DX,"dY = ";DY:GOTO 100
```

Plug a joystick into the player #1 slot, then run the program. Observe the DX and DY values for each stick position.

Now, try the short demo program below. Study this program to see how the joystick routine can be used in your programs.

## PROGRAM. Atari BASIC Joystick Routine.

```
1 REM .....SCRIBBLE DEMO PROGRAM
10 GRAPHICS 23:X=79:Y=47:COLOR 1:GOTO 160
20 S=STICK(S)
30 DX=(S=5 OR S=6 OR S=7)-(S=9 OR S=10 OR S=
   11)
40 DY=(S=5 OR S=9 OR S=13)-(S=6 OR S=10 OR S
   =14)
50 RETURN
100 S=0:GOSUB 20
110 IF  NOT (DX OR DY) THEN 100
120 X=X+DX:IF X>159 THEN X=0
130 IF X<0 THEN X=159
140 Y=Y+DY:IF Y>95 THEN Y=0
150 IF Y<0 THEN Y=95
160 PLOT X,Y
170 N=255-INT((X/159+Y/95)*125)
180 SOUND 0,N,10,8
190 GOTO 100
```

# Joystick Tester ✗

Robert Rochon

*This short routine permits you to easily test your joysticks and pinpoint potential problems.*

The Atari joysticks are mass-produced and tend to malfunction. I have seven joysticks and only three are working properly. The problems are in the contacts which are held together with tape which loosens much too easily. What a pity; the general design of the joysticks is very good.

This program quickly reveals any flaws in a joystick. You should be able to make the screen red when the fire button is pressed. You should also be able to make nine blue dots for the nine different joystick positions. To clear the screen, press any key.

You can also find out how sensitive a joystick is by controlling it with one finger only. With this test a sticky joystick will show up like a sore thumb. For easy identification, number your best joystick. I always use my most sensitive joystick for Star Raiders. Hint: Jamming a ¾ inch plastic, T-shaped pipe connector onto the stick of the joystick will make it twice as sensitive.

## Program Design

|  |  |
|---|---|
| 2 | turns on GR.3, turns off cursor (POKE 752). |
| 5-15 | records joystick position. |
| 200 | erases old line and draws new line. |
| 230 | plots center and outside dot. |
| 300 | colors the screen red when fire button is pressed. |
| 500-520 | writes the value of STICK(0) AND STRIG(0); PEEKs 656 & 657 keeps the writing in one place. |
| 540 | Keep the screen when any key is pressed. |
| 999 | C & D last joystick position (C & D are used to minimize blinking on the screen.) A & B new joystick position creates continuous loop. |

**PROGRAM. Joystick Tester.**

```
2 GRAPHICS 3:POKE 752,1:GOTO 999
5 A=24:B=14:GOTO 200
6 A=24:B=6:GOTO 200
7 A=25:B=10:GOTO 200
9 A=16:B=14:GOTO 200
10 A=16:B=6:GOTO 200
11 A=15:B=10:GOTO 200
13 A=20:B=15:GOTO 200
14 A=20:B=5:GOTO 200
15 A=20:B=10
200 IF C<>A OR D<>B THEN COLOR 4:DRAWTO 20,1
    0:COLOR 2:DRAWTO A,B
230 COLOR 3:PLOT 20,10:PLOT A,B
300 POKE 712,66-STRIG(0)*66
500 POKE 656,1:POKE 657,5
510 ? "STICK[illegible]";STICK(0),
520 ? "STRIG[illegible]";STRIG(0)
540 IF PEEK(764)<>255 THEN POKE 764,255:RUN
999 C=A:D=B:GOTO STICK(0)
```

# Keyboard Input Or Controlled Escape

Brian Van Cleve

*The BASIC INPUT statement is prone to user error. What is needed is controlled input, where each key is checked for validity. The following program even checks for the START key to permit an "escape" function.*

Here is a short subroutine that I use in all my menu driven programs. It allows the user to enter data as usual while checking for the start key being pressed. If it discovers use of the start key, the subroutine provides a controlled escape.

To use this, DIM IN$ to the maximum expected input and open the keyboard for input using OPEN #1,4,0, "K:". Set the variable KEY to the first line number in the subroutine and the variable ESC to the line to return to if the user presses the START key (like the main menu, for example). Then use a GOSUB KEY for any input. The user input will be returned in IN$.

Now for the program.

## PROGRAM. Keyboard Input

```
1000 REM KEYBOARD               I use REM as the first statement in
   INPUT                           my subroutines, easy to move
1005 IN$=""                     Set IN$ to null
1010 POKE 764,255               Clear keyboard buffer out
1020 POKE 756,224               Force uppercase letters
1030 POKE 694,0                 NO inverse video
1050 IF PEEK(53279)=6           START key is pressed
   THEN POP :GOTO ESC           Pop the return off stack and esc
1060 IF PEEK(764)=255           No key pressed yet
   THEN 1050
1070 GET #1,K                   Key pressed, get it
1080 IF K=155 THEN              Pressed return key
   RETURN
1090 IN$(LEN(IN$)+1)            Put key in string
   =CHR$(K)
1100 ? CHR$(K);                 Print it on screen
1110 GOTO 1050                  Loop til done
```

# POKE TAB In BASIC

Lawrence R. Stark

*A common problem, the lack of a TAB statement in Atari BASIC, is solved with a single POKE.*

Perhaps my first experience with the Atari was similar to yours. Looking at the machine on display and generally feeling favorable, I finally pushed aside a few space war players and sat down to try it out. Having found the BASIC ROM cartridge, I tried to construct a short test program. The only program that came to mind in this stage was to do a few loops and GOSUBs with the phrase: "THIS IS A TEST." To vary it a bit, I fooled around with print positions.

But do this in Atari BASIC and you discover that the TAB command is not there. So you make do with spaces and the like to design printed output. This does not work well. A few subroutines, including some published in **COMPUTE!**, can be of help, but can also be awkward.

## It's In The Maps

So finally I started looking at memory map listings. And there it was. The map is full of TABs. They are at locations 82,85,91, and 657. All one has to do is POKE in new values and most of these work – on the TV screen. Not much happens on the printer.

But what is this at location 201? PTABW is the name. It even looks like a shortened name of Print-Tab-Width. And it is! It is the missing TAB command, and in this case it works for both printer and TV screen.

Experiment with it a bit and you learn its rules. First, its formula is POKE 201, nn, followed by PRINT, avar or svar. Remember the comma; after all, it is the tabulator that you are modifying.

But there are two hitches. One is that if you try it with nn at 0, awful things happen. A loop is entered and even the BREAK key doesn't interrupt. The second limitation is that the default setting is not restored except on powering down the computer. Even RESET does not return the value at 201 to 10.

When using the Atari "TAB" in a program, you will have to provide error messages or traps and also reset the default setting. The demonstration program shows an example. The value of nn is tested for "less than one" and a "bypass" is executed if it is. This could just as easily work like an ordinary error message if you have the program STOP or END on less than one. The value at 201 is restored before the program shuts off at line 800. The demonstration program is what might be called the "page centering algorithm." It is used here because it is illustrative of several features of the pseudo-TAB in Atari BASIC.

## PROGRAM. POKE TAB In BASIC.

```
10 DIM A$(38):? CHR$(125)
15 TRAP 800:REM Trap end of data
20 READ A$
50 X=17-(LEN(A$)/2):REM Center at 17 on 38 c
   ol TV screen
60 IF X<1 THEN 20:REM Bypass a potential cra
   sh
100 POKE 201,X
120 ? ,A$
150 GOTO 20
800 POKE 84,20:POKE 201,10:END :REM for a ne
    ater screen AND to reset default tabs
899 REM Commas in DATA statements for blank
    lines
900 DATA A TALE OF TWO CITIES,,A Novel,,,,by
    ,,Charles Dickens,,,,,,,,1832
```

# The 49 Second Screen Dump

David Newcorn

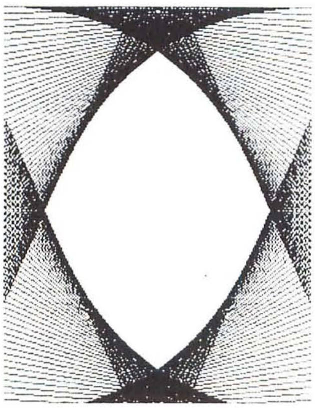
*A machine language routine and a BASIC loader you can append to any graphics program to dump the contents of your screen to an Epson MX-80 printer (with Graftrax) ... in only 49 seconds.*

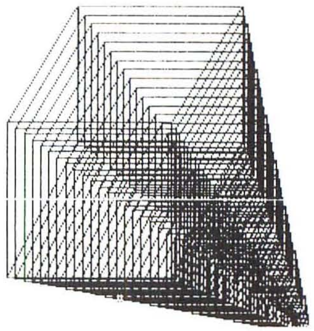
If you have an Epson MX-80 printer with Graftrax and an Atari 800 with a beautiful Graphics 8 picture on your screen, then what do you do? Dump it to the printer, of course! How? At first I dumped my pictures to the printer through BASIC by scanning each dot separately with the LOCATE statement. It took about thirty minutes to do one picture. Realizing that this was crazy, I redid the program so it would PEEK into display memory and grab eight pixels (one byte of display memory) at a time instead of only one pixel at a time. This reduced printing time to about four and a half minutes, which is still quite slow. So then I turned to machine language. The routine that follows dumps your Graphics 8 picture to your printer in only 49 seconds! Sure beats tying up your computer for half an hour and waiting around for your picture to be printed.

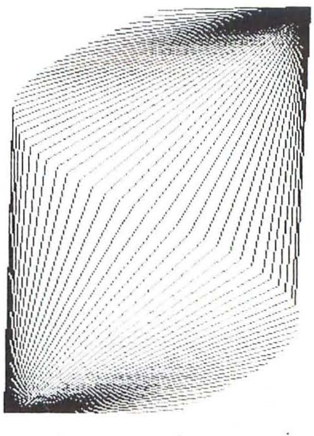
## How It Works

Before I explain how it works, let me refresh your memory on how display memory is organized in Graphics 8. Every eight pixels forms one byte of display memory. So, positions 0,0 through 7,0 would be one byte, and positions 8,0 through 15,0 would be another byte, and so on for 38 more bytes across the screen. The BASIC routine sets up a loop which scans the X axis of display memory, starting with the left byte, and finishing with the 40th byte on the right of the screen (0 to 39 is 40). During each row, control is passed to the machine language subroutine along with the address of the string where the result is to be stored and the base address of the column of display memory to be dumped

The machine language program starts at the bottom of the screen and scans eight bits at a time all the way to the top. After it looks at the current display memory byte, it stores it in the current string address. It then increments the string address and decrements the display memory address by subtracting 40 to move up to the next line. (The display memory is one-dimensional, and the display screen is two-dimensional. That's why it subtracts 40 to move up to the next line. It goes back 40 bytes in memory.) When it finishes at the top of the screen, control is passed back to BASIC. BASIC then prints the string to the printer where each character represents a byte of screen memory. The bit pattern of the character is mapped directly to the printhead. After the string is printed, the current column is incremented in BASIC to pass along to the USeR function for the next go-around. Simple, right? Right. Just append Program 1 to your program which draws the picture, and after it finishes, use a GOTO or GOSUB to this routine. That's all you have to do. Just sit back and enjoy your 49-second picture.







## PROGRAM 1. The 49 Second Screen Dump.

```
500 DIM A$(192):FOR B=1 TO 61:READ N:POKE 15
    35+B,N:NEXT B:DM=PEEK(88)+PEEK(89)*256:D
    M=DM+40*191
505 REM POKE IN M/L PROGRAM AND SET UP DISPL
    AY MEMORY POINTER
510 LPRINT CHR$(27);"A";CHR$(8):FOR X=DM TO
    DM+39
515 REM SET LINE SPACING AND MAKE LOOP
520 A$=CHR$(0):A$(192)=CHR$(0):A$(2)=A$
540 W=USR(1536,X,ADR(A$)):LPRINT CHR$(27);"K
    ";CHR$(192);CHR$(0);A$
545 REM PASS BOTH VALUES TO M/L PROGRAM, AND
     PRINT STRING
550 NEXT X
560 DATA 104,104,141,21,6,104,141,20,6,104,1
    41,27,6,104,141,26,6,160,193,173,255,255
    ,136,240,35,141,255,255,238
570 DATA 26,6,240,21,173,20,6,56,233,40,141,
    20,6,144,4,24,76,19,6,206,21,6,76,19,6,2
    38,27,6,76,33,6,96
```

## PROGRAM 2. The 49 Second Screen Dump.

```
10 ;ATARI 800 SCREEN DUMP UTILITY FOR DUMPIN
   G GRAPHICS 8 PICTURES
20 ;TO EPSON MX-80 PRINTERS WITH GRAFTRAX.
30 ;BY DAVID NEWCORN 2/28/82
40 ;ASSEMBLY LANGUAGE LISTING
0100 ADR    =     $FFFF      ;DUMMY ADDRESS (S
    CREEN MEM ADDR)
0110 STR    =     $FFFF      ;DUMMY ADDRESS (S
    TRING ADDR)
0120        *=    $600
0130        PLA              ;PULL OFF AUX BYT
    E FROM BASIC
0140        PLA              ;PULL HI BYTE OF
    STRING STORAGE
0150        STA   LOA+2      ;STORE HI BYTE
0160        PLA              ;PULL LO BYTE OF
    STRING STORAGE
0170        STA   LOA+1      ;STORE LO BYTE
0180        PLA              ;PULL HI BYTE OF
    BEGINNING OF SCREEN MEM
0190        STA   STO+2      ;STORE HI BYTE
0200        PLA              ;PULL LO BYTE OF
    SCREEN MEM
0210        STA   STO+1      ;STORE LO BYTE
0220        LDY   #193       ;LOAD Y AXIS COUN
```

```
    TER
0230 LOA
0240          LDA   ADR          ;LOAD SCREEN BYTE
0250          DEY                ;DECREMENT COUNTE
    R
0260          BEQ   RET          ;IF DONE THEN RET
    URN TO BASIC
0270 STO
0280          STA   STR          ;STORE SCREEN BYT
    E IN A$
0290          INC   STO+1        ;INCREMENT LOW EN
    D OF STRING
0300          BEQ   BIG1         ;IF LOW END OVERF
    LOWS, THEN INC HI END
0310 CONT
0320          LDA   LOA+1        ;LOAD ACCUM WITH
    LOW SCREEN ADDRESS
0330          SEC                ;SET CARRY BIT FO
    R SUBTRACT W/O BORROW
0340          SBC   #40          ;SUBTRACT 40 (40
    BYTES PER SCAN LINE)
0350          STA   LOA+1        ;STORE RESULT
0360          BCC   BIG          ;IF UNDERFLOW, DE
    C HI BYTE OF SCRN MEM
0370          CLC                ;CLEAR CARRY
0380          JMP   LOA          ;LOAD NEXT BYTE
0390 BIG
0400          DEC   LOA+2        ;DECREMENT HI BYT
    E OF SCREEN MEM
0410          JMP   LOA          ;LOAD NEXT BYTE
0420 BIG1
0430          INC   STO+2        ;INCREMENT HI BYT
    E OF STRING STORAGE
0440          JMP   CONT         ;CONTINUE
0450 RET
0460          RTS                ;RETURN TO BASIC
```

# Memory Test

Ed Stewart

All machines sometimes break and must then be fixed. My Atari computer is no exception to this rule. The difficulty lies in determining what went wrong. As you probably know, it's sometimes difficult to ascertain the cause of an error. The first question that should be asked is this:

## Is The Problem Hardware Or Software?

This is sometimes very difficult to answer, but should be done before you shell out fifty bucks for someone to look at your Atari. In this article I will provide you with one helpful tool that you can use to help answer this question if your Atari goes off the deep end some day.

One of the most critical resources in any computer system is the dynamic memory known as RAM. This memory, unlike disk or tape memory, is operated upon directly by your Atari CPU, the 6502. Information on disk or tape memory devices must be first placed into RAM memory before it can be used either as a program or by a program. If an error were to occur at any RAM location where you have a program or data stored, the program would probably not function correctly. The error encountered could result in "lock up" or practically any other conceivable symptom. It could even make you think that the program is at fault if other programs appear to function normally. The reason some programs may function o.k. is that they do not reference the particular RAM location that is in error. Only one faulty bit in one byte may be bad, but this would be enough to do the trick. Computers are generally not tolerant of data errors, especially in their RAMs. Most large scale computers have error checking circuitry built into RAM memory, but this has not been done for our Atari friend. The new IBM personal computer has such RAM checking circuitry, so it is probably only a matter of time until this becomes commonplace for all our microcomputers. For now, though, the question must be:

## How Can I Check My RAM?

I'm glad you asked that question, because it just so happens that I have a program to do just that. Of course, if you wish to buy a RAM test program you may still do that, but you should find this program as helpful as any available for most of your needs.

## About The Program

This program consists of a BASIC program that has the assistance of a little machine language program. The BASIC program determines the amount of free RAM that can be used for testing purposes and asks you how much of that RAM you wish to actually test. It then proceeds to test that memory range you specify by repetitively invoking the Machine Language Program (MLP). The MLP tests every bit in every byte of the requested range by testing every number capable of being represented in the RAM area. After the area is tested, and if there are no errors, return is made to the BASIC program, with a successful completion indicated. If an error is encountered in RAM by the MLP, then return is made to the BASIC program, indicating the particular problem found. The BASIC program will stop execution if an error is found and will display error information on the display screen. To continue execution after an error is found, you must depress any key on the keyboard. The testing will then continue with all subsequent bytes in the RAM area found to be defective.

During normal execution of this program the display screen will appear blank. This is done to speed the testing process. The program will provide an audio signal after each testing pass is completed, but the screen will still remain blank. If you wish to see what pass the test is on, just depress any key and the program will pause with a good display turned on after the current pass is completed. Depressing a key again will cause the program to continue. If you wish to test a different RAM area, you can press BREAK or SYSTEM RESET at any time, followed by RUN. To improve the speed of this program requires that the amount of memory to test must be at least 256 bytes.

The BASIC program is fairly easy to follow and is documented with REM statements. You may remove all REM statements to get back a little more RAM. The MLP is included as DATA statements in the BASIC program.

## No Errors Found

If the program finds no errors in the RAM you are testing, this does not mean that RAM is necessarily free of all errors, as some RAM will not be tested. RAM in locations HEX 0-6FF cannot be tested, and neither can display list RAM or the RAM occupied by this program, because this testing program only tests "unused" RAM. If your machine is 8K or 16K, or if you have a 400, then you cannot do what I am about to recommend, although you can rest assured that you have been able to test most of your RAM memory. For all you lucky 800 owners with more than one memory board, it is now time to rearrange your RAM boards if you have that ability. By changing the position of your RAM boards you will be able to test a greater portion of RAM memory. If, for instance, you have three 16K boards, you can be assured that the entire 16K will be tested for the middle board, although the first and third boards will contain some RAM that can't be tested until it is placed in the middle slot. If you still find no errors after testing all of RAM, then your problem is not with the RAM. Great, huh?

## An Error Found

If the program stops with an error encountered in RAM, then you have probably found the source of the error. Usually through rearranging the RAM boards the characteristics of the problem will change, but the RAM will still be faulty. You now have a decision to make. You can either remove the faulty board and fix it yourself, or you can have someone do it for you. If someone does it for you, give them the results of your RAM test. The repair should be much easier for them and, possibly, somewhat less costly. You may even luck out and catch a RAM board still on warranty.

## PROGRAM. Memory Test.

```
51 REM ATARI RAM TEST PROGRAM
52 REM BY ED STEWART 03/82
53 REM 11025 SAGEBRUSH AVE
54 REM UNIONTOWN OHIO 44685
99 REM SETUP SOME REQUIRED CONSTANTS
100 N1=1:N2=N1+N1:N255=255:N256=N255+N1
200 DIM S$(N2):S$(N1,N1)=CHR$(157):S$(N2,N2)
    =CHR$(159)
299 REM READ IN THE MACHINE LANGUAGE PROGRAM
300 GOSUB 2900
399 REM GET LOW AND HIGH MEMORY BOUNDS
400 L=PEEK(15)*N256:H=PEEK(742)*N256:IF PEEK
    (14)<>NO THEN L=L+N256
499 REM DISPLAY BOUNDS AND GET REPLY
500 ? CHR$(125);S$;"ATARI MEMORY TEST PROGRA
    M";CHR$(155);S$;"MEMORY BOUNDS ARE"
600 ? S$;"LOW=";L:? S$;"HIGH=";H
700 ? S$;"GIVE TEST BOUNDS";CHR$(155)
800 TRAP 800:? S$;"LOW=";:INPUT LOW:IF LOW<L
     OR LOW>H THEN 800
900 TRAP 900:? S$;"HIGH=";:INPUT HIGH:IF HIG
    H>H OR HIGH<L OR HIGH-LOW<N256 THEN 900
999 REM SETUP BOUNDS FOR THE MLP
1000 POKE 205,NO:POKE 206,INT(HIGH/N256)
1100 TRAP 32767:POKE 203,NO:POKE 204,INT(LOW
     /N256)
1200 POKE 764,N255
1299 REM INVOKE THE MLP TO DO THE TEST
1300 POKE 559,NO:POKE 764,N255:X=USR(1536)
1399 REM CHECK RETURN FROM MLP
1400 IF PEEK(208)=NO THEN 2200
1499 REM SHOW MEMORY ERROR ON SCREEN
1500 ? " ERROR AT ";(PEEK(203)+PEEK(204)*N25
     6);" EXP=";PEEK(207);" ACT=";PEEK(209)
1600 SOUND NO,PASS,6,8:FOR I=N1 TO 5:NEXT I:
     SOUND NO,NO,NO,NO
1699 REM SETUP NEXT BYTE TO TEST SO WE DONT
     STOP WITH FIRST ERROR
1700 IF PEEK(203)=N255 THEN POKE 204,(PEEK(2
     04)+N1):POKE 203,NO:GOTO 1900
1800 POKE 203,(PEEK(203)+N1)
1900 POKE 764,N255:POKE 559,34
1999 REM CONTINUE ONLY IF KEY PRESSED
2000 IF PEEK(764)=N255 THEN 2000
2099 REM CONTINUE TESTING BAD RANGE
2100 GOTO 1300
2199 REM GOOD TEST PASS SO SAY SO
2200 PASS=PASS+N1:? " GOOD PASS NUMBER ";PAS
```

```
     S:SOUND NO,PASS,10,8
2300 FOR I=N1 TO 5:NEXT I:SOUND NO,NO,NO,NO
2399 REM STOP AND DISPLAY STUFF IF KEY IS PR
     ESSED
2400 IF PEEK(764)<>N255 THEN 2600
2499 REM CONTINUE WITH NEXT PASS
2500 GOTO 1100
2600 POKE 764,N255
2699 REM WAIT HERE UNTIL A KEY IS PRESSED
2700 POKE 559,34:IF PEEK(764)=N255 THEN 2700
2799 REM CONTINUE WITH NEXT PASS
2800 GOTO 1100
2899 REM READ IN MACHINE LANGUAGE PROGRAM
2900 FOR L=1536 TO 1576:READ H:POKE L,H:NEXT
      L:RETURN
3000 DATA 104,169,0,160,0,24,145,203,209,203
     ,208,18,105,1,208,246,200,208,242,230,2
     04,166,204,228,206
3100 DATA 208,234,133,208,96,133,207,177,203
     ,133,209,169,1,133,208,96
```

CHAPTER TWO

# PROGRAMMING TECHNIQUES

# Atari BASIC String Manipulation Tricks

David E. Carew

*While the merits of Atari's method of handling strings have been the subject of some debate, David Carew suggests how to make Atari strings work effectively. He provides some interesting techniques.*

BASIC programming languages were influenced, naturally enough, by the minicomputer BASIC languages which preceded microBASIC's. The most popular of these minicomputer BASICs was probably Digital Equipment Corporation's (DEC's) PDP-11 family of BASICs: BASIC-11, BASIC-Plus and BASIC-Plus2. These BASICs possessed (and still possess) special substring manipulation functions such as MID$, LEFT$ and RIGHT$, and implemented "arrays of strings" which were referenced via subscripts exactly like a numeric array. Microsoft BASIC is a child of such ancestry. Other large vendors, including Data General Corporation, used the subscript syntax of A$(X,Y) to handle substrings, which eliminated special syntax (MID$,*et al*) for substring functions while precluding a "nice" implementation of string arrays. Atari's BASIC is one of this type. The DEC/Microsoft approach may be more popular than the DG/Atari approach – so much so that it is sometimes necessary to remind ourselves that simpler syntax does not necessarily mean inherently less power. Indeed, a simpler BASIC syntax may well mean that the BASIC interpreter uses fewer hardware resources, leaving more for our programs. Some different coding techniques are definitely called for, however.

I have a few tricks up my sleeve. I present some of these here in order to open up the possibilities to you.

The idea is to play to the strengths of the tool you have. One outstanding strength of Atari BASIC is its capability of addressing very long strings. What can you do with very long strings? Well, who says you can't build a viable word processor or editor in BASIC when you have the power to control and manipulate an edit buffer as a single huge string of characters? One can initialize a long string of an inconvenient large size, without inconvenient large string literal statements, and without a large number of iterations by concatenating the string to itself:

```
1000 E$=" ":TRAP 1010:FOR J1=1 TO 15:E$(LEN(
     E$)+1)=E$
1010 NEXT J1
```

For data base applications, a generalized "screen forms" handler might build and store desired screen input/display formats as a long "string image." One can precisely and flexibly embed particular string information into a surrounding string image with just one statement:

```
1200 REC$(J+1-LEN(EX$),J)=EX$
```

The above places the right edge of EX$ at position J in REC$, effectively right justifying variable-length information into REC$. To accomplish a similar left justification, place the left edge of EX$ at position J in REC$:

```
1300 REC$(J,J-1+LEN(EX$))=EX$
```

If we begin using long strings as the handy data structures that they are, then it will be occasionally necessary to "pad" a piece of data with blanks so that it precisely fits a "longstring" subfield. Adding the correct number of blanks to short data can be accomplished in a single line like this:

```
1400 LN=LEN(EX$):IF LN<25 THEN EX$(LN+1)=BLA
     NK$(1,25-LN)
```

This code shows EX$ being padded with "trailing" blanks to a length of 25 characters. EX$ must, of course, be shorter than 25 characters in the first place, and BLANK$ must be initialized and left as a string of blanks. Change the literal 25 to a variable, set it as necessary, and the same single line in a subroutine will pad any string to any required length.

Stripping the blanks from data taken out of a long string is also a necessary housekeeping chore. The code to accomplish this is again very compact, but it does involve iteration (a "loop"):

```
1500 LN=LEN(EX$):IF LN>1 AND EX$(LN,LN)=" "
     THEN EX$=EX$(1,LN-1):GOTO 1500
```

Line 1500 strips away "trailing" blanks on the right end of EX$. Any "leading" blanks at the left end of EX$ are just as vulnerable to an analogous technique:

```
1600 IF EX$(1,1)=" " THEN EX$=EX$(2,LEN(EX$))
```

```
:IF LEN(EX$)>=2 THEN 1600
```

Substituting a variable equal to a blank for the literal blank in the above code will speed up execution; the literal, of course, improves readability. The choice in this tradeoff is yours to make.

Long strings have much inherent power, and the possibilities are endless and exciting. The old micro trick of storing graphics and machine language routines as BASIC strings and/or string literals is made all the more attractive by this power. And the tantalizing prospect of "programs that write programs" absolutely cries out for long strings that turn out to be program code. The substring manipulation techniques to be used are not as obvious without the special function calls of other BASICs. However, having seen a few such techniques in this article, you will not, I trust, now ignore Atari BASIC's "longstring" power, when this power could be of use to your application.

# Using The Atari Forced Read Mode

Frank C. Jones

*Automatic data entry, line numbering, deletion, self-modifying programs – this programming method opens up a new level of control over the computer's behavior during a RUN.*

There are many occasions when it would be useful if one could cause data and program lines to be entered into the computer's memory without having to push the RETURN key. Such a facility could be used to enable a program to alter itself by adding or deleting program lines or to automatically reenter data that had been temporarily stored on the screen. In the article "Restoring and Updating Data on the Atari" (**COMPUTE!** August, 1981, #15) Bruce Frumker has shown that the Atari computer does indeed have this capability. In this article we will explore this facet of the Atari a little further with an eye to gaining a better understanding of how it works, and give a few examples of how it can be put to good use.

To begin our exploration we must visit those often mentioned, but little understood, objects – the IOCB's. IOCB stands for Input/Output Control Block. There are eight of them, and each one is nothing more than sixteen contiguous bytes of RAM. IOCB #0 runs from $340 to $34F (Dec. 832 to 847), IOCB #1 runs from $350 to $35F (Dec. 848 to 863), and so on.

Most input and output is handled by a portion of the Atari Operating System called the CIO (Central Input/Output) facility. When the CIO is called by the program that is ready to do some I/O, only one number is passed to it, the number of the IOCB that is to control the actual I/O operation. All of the other information that the CIO needs (to perform the required I/O operation for you purists) is contained in the sixteen bytes of the IOCB in question. Of course, all of the required information must have been placed in the proper bytes of the IOCB prior to the jump to the CIO entry point. One of the ways that such information is placed in the IOCB is with the OPEN statement in BASIC. However, we are getting a bit ahead of ourselves; we will return to the OPEN statement shortly.

A few examples of the sort of information that the CIO will find in the IOCB are: the first byte contains the ID number of the device that is to exchange data with the computer, the third byte contains the code that tells CIO what it is supposed to do (read data, write data, etc.), the fifth and sixth bytes contain the address of the data to be output or the location where incoming data is to be stored, and so on until the eleventh byte (the auxiliary information byte or ICAX1).

Here is where we return to the OPEN statement of BASIC. When a BASIC program executes a statement such as OPEN #2,8,0,"P", IOCB #2 is set up to write data to the printer. The number 8 in the command means "write data." What many do not know is that the OPEN command sends that number 8 to the eleventh byte of IOCB #2 or, in other words, ICAX1. The number 8 is represented by the 3 bit of ICAX1 being set. This tells CIO that the channel has been set up for output. If the 2 bit were set, that sould represent number 4 and would mean that the channel was set up for input. If both bits are set (a number 12, output and input are both enabled.

We can see now why the 2 and 3 bits of ICAX1 are called "direction bits" – they control the direction of data flow. However, this is not all that the bits of ICAX1 can do. Certain I/O devices can be made to do special things by setting some of the other bits. The Screen Editor is one of these certain devices. It supports a mode of operation that Atari calls the *force read mode.* It would seem to me that "forced enter mode" is a more descriptive name, but I suppose that this sounds too much like "forced entry" and hence, a bit too felonious. This mode is enabled by setting the zero bit of ICAX1 in any IOCB that is OPENed to the Screen Editor. Setting the zero bit is, of course, accomplished by adding one to the number that the OPEN command sends to ICAX1.

This means that a command such as OPEN #2,5,0,"E" or OPEN #2,13,0,"E" would set up channel number 2 to the Screen Editor in the forced read mode. Note that OPEN #2,9,0,"E" is not appropriate since it doesn't make much sense to invoke the forced read mode in a channel that has been set up for output only.

The obvious question that now comes to mind is: What is the forced read mode? The answer is simple. Whenever an INPUT command is issued over a channel (IOCB) that has been set up for forced read, the Operating System does not wait for the operator to push the RETURN key, but immediately INPUTs the data from the logical line in which the cursor is residing. *All data that is under or to the right of the cursor may be INPUT at this time.* Note carefully this last remark. This mode does not work precisely like the more usual one in which data to the left of the cursor may be INPUT.

As an example of the utility of this mode we will now write the world's simplest text screen dump. Early in the program the forced read mode should be set up with a program line such as:

```
0 OPEN #2,5,0,"E":DIM LINE$(120):SDUMP=32000
```

Since OPENing an IOCB to the screen editor clears the screen, this line should be executed *before* anything is written to the screen that you might want to dump to the printer. LINE$ is the string that will hold a line of text from the screen and SDUMP is the line number of the beginning of the dump routine. It can be called from anywhere in your program or from the immediate mode by the command GOSUB SDUMP. The dump routine is as follows:

```
32000 POSITION PEEK(82),0
32010 FOR I=1 TO 24
32020 INPUT #2,LINE$:LPRINT LINE$
32030 NEXT I:RETURN
```

What could be simpler? Of course, this won't format the text on the printer exactly as it appears on the screen, and it might cause some scrolling if you have some logical lines that are longer than one physical line, but it *will* dump all of the text on the screen to the printed page.

There are many other cases in which it is advantageous to be able to read data from the screen automatically. One occurred to me as I was trying to work around what seemed at the time to be an insurmountable problem. I was working on a program that was to read tape files and perform calculations on each file before going on to the next one. A fairly large array was needed whose size depended on the data in each file. I could DIMension the array to the largest size that I would ever need, but that would be wasteful of memory. Besides, I wasn't sure that I could ever guess just how large that would be. An alternative would be to use the CLR command and reDIMension the array after reading each data file.

The trouble with that is that I wanted to maintain a running total from one file to the next and the CLR command would clear *all* variables. While pondering this dilemma, I decided that I could write the current value of the running total on the screen, use the CLR command, and then read it right back again using the forced read mode. Of course, if you have a disk system all of this is unnecessary; you can use a disk file as temporary storage. But if (like me) you have only cassette storage, this kind of file manipulation is not possible. It is nice that folks like us do have the alternative of the forced read mode.

This method is by no means limited to storing only one value; you can print several numbers, separated by commas with a command like PRINT A;",";B;",";C; – etc. and then read them back in with a multiple variable INPUT statement. If you have reasons for not wanting all of this screen activity to be seen, you can make the writing and background the same color, as Bruce Frumker did. Or you can disable screen DMA with a POKE 559,0, then do your thing with the screen. When you are done, clear it, and restore DMA with a POKE 559,34. I am still thinking of variations on this technique, and I am sure that you will think of many that have not occurred to me.

So far none of this allows a BASIC program to modify itself in any way, although Bruce Frumker's mysterious POKE 842,13 should be taking on a familiar look. A little calculation will show you that location 842 is the ICAX1 byte of IOCB #0, the one IOCB that BASIC won't let you OPEN, CLOSE or otherwise modify. The reason that BASIC won't touch IOCB #0, is that it is reserved for the Operating System to use in communicating with the screen editor. Whenever a BASIC command such as PRINT, ?, LIST, or INPUT that does not specify a particular IOCB is executed, the Operating System uses IOCB #0 to do the job. In fact, any time you use the screen editor to communicate with the Atari, such as entering immediate mode commands, entering or deleting program lines or anything, you do it through IOCB #0.

Even when it may look as if nothing at all is going on with your Atari, the Operating System is actually issuing repeated INPUT (or its equivalent) commands through IOCB #0 to the screen editor. It is just waiting for you to print something on the screen and then press RETURN. Of course, if IOCB #0 were in the forced read mode it wouldn't wait for you to press RETURN, would it? The idea is getting closer (although I'll bet that a lot of you are already there).

Although the general rule exists that once ICAX1 has been set by an OPEN command it shouldn't be changed, it turns out that turning the forced read mode on and off is an exception and the statements POKE 842,13 and POKE 842,12 do just that. Once IOCB #0 has been placed in the forced read mode and the BASIC program relinquishes control to the Operating System (with a STOP or END statement), the Operating System starts immediately scanning the screen for program lines to add or delete or immediate mode commands to carry out. This will continue until it comes across a command that returns control to the BASIC program (CONT RUN, or GOTO line#) or turns off the forced read mode (POKE 842,12).

Although this may sound very straightforward, it takes some careful planning to position the cursor and the printed lines to make sure that the desired lines are, in fact, read when the Operating System takes over. A few things need to be remembered:

● First, when a program line (including a blank line) is read, the cursor jumps to the beginning of the next logical line to INPUT the next line. So far, so good.

● When a BASIC program comes to a STOP statement, the cursor skips a line, prints "STOPPED ON LINE line#," and is ready to read on the line following that. In other words, you must position the cursor two lines above the line where you wish to start reading. (Using END to stop the program does the same thing, only READY is printed instead.)

● When an immediate mode command is read, the cursor skips a line, prints READY and starts reading again on the line following that; i.e., two lines are skipped.

If the lines to be read are not positioned with these facts in mind, some of the lines may be missed and not read. This can be especially annoying when the line that is missed is the one that turns control back to the BASIC program or turns off the forced read mode. In this case the cursor "runs away" and continues to look for lines to read until something stops it. There is no need to panic, however. The SYSTEM RESET key will bring everything back to normal.

A little practice is all that it takes to get the hang of setting up the screen to be read. If you find that some of your lines are not being read or the cursor is running away, look for something that is making the cursor skip a line or two that you had not counted on.

Uses for this mode abound. Have you ever wondered how an algebraic formula could be entered into a program at run time and become incorporated in the program to be evaluated and plotted? The forced read mode is the answer. In fact, Atari programmers have used just this technique in the GRAPH IT programs. A program that initially POKEs a long machine language subroutine into memory when it is first RUN can eliminate this POKEing routine when it is through with it. In this way, it can avoid this time consuming and unnecessary process when it is reRUN.

The automatic generation of DATA statements as in Bruce Frumker's program is a natural use of the forced read mode. However, there is one case where it is almost a necessity to remove this function from error prone human hands. Anyone that has used the Atari Editor/Assembler cartridge to write a machine language subroutine has had to face the rather tedious job of incorporating it into the BASIC program that is to use it. This job is not only difficult, but is a very likely source of errors as it is very hard to type a meaningless string of numbers without making at least one mistake, and machine language is not at all forgiving of the smallest mistake.

Getting the program in memory in the first place is not too difficult using the short BASIC program supplied with the Editor/ Assembler manual errata sheet. The real problem comes when one wants to integrate it into the BASIC program. Program 1 should be a help in this respect. It reads the binary file generated by the Editor/ Assembler SAVE command and generates the appropriate DATA statements for subsequent POKEing into memory.

In the program you will see a couple of words in curly brackets. They represent screen control characters. They are printed by pressing the ESC key and the appropriate control key. (CLEAR) is the screen clear character, and (BACK) is the DELETE BACKSPACE key. Now for the program description.

**Line 5** – Get ready for the EOF record

**Line 10** – This is written for cassette files; the generalization to any kind of file is obvious

**Lines 20-40** – The first six bytes of a binary file contain the location in RAM of the program; it is assumed that the BASIC program will POKE it into the correct location

**Line 50** – The J's are the DATA statement line numbers; they may be changed at your convenience (except ›460). Be sure to have enough.

**Line 70** – The range of I has been chosen to put twenty-five numbers in each DATA statement. This, too, can be changed, but you must make sure not to exceed a logical line length

**Line 95** – Erases the last comma

**Line 100** – Prints the command to return control to the program

**Line 110** – Positions the cursor, turns on the forced read r..ode, and turns control over to the Operating System

**Line 120** – Turns off the forced read mode

**Line 300** – Checks the error number and if it is an EOF, lines 400-430 set up the last DATA statement and enter it into memory

**Lines 440-450** – Clear the screen and LIST the data statements

When this program has finished its job, the DATA statements can be listed to a file for later inclusion in a BASIC program. It should be easy to modify this program to your own needs. The basic framework is there, and now that you understand how it works, write one that fits your programming style more closely.

I am sure that these uses of the forced read mode are only the first hints of what will become an entirely new dimension in Atari programming. I am eager to see what will come next.

**PROGRAM. Using The Atari Forced Read Mode.**

```
5 TRAP 300
10 OPEN #1,4,0,"C"
20 FOR I=1 TO 6
30 GET #1,A
40 NEXT I
50 FOR J=500 TO 5000 STEP 10
55 PRINT "{CLEAR}"
60 POSITION PEEK(82),2:PRINT :PRINT J;" DATA
   ";
70 FOR I=1 TO 25
80 GET #1,A:PRINT A;",";
90 NEXT I
95 PRINT "{BACK S}"
100 PRINT "CONT"
110 POSITION PEEK(82),0:POKE 842,13:STOP
120 POKE 842,12
130 NEXT J
300 IF PEEK(195)=136 THEN 400
310 PRINT " ERROR - ";PEEK(195):END
400 PRINT "{BACK S}"
410 PRINT "CONT"
420 POSITION PEEK(82),0:POKE 842,13:STOP
430 POKE 842,12
440 PRINT "{CLEAR}"
450 LIST 500,5000
460 END
```

# A Simple A Screen Editor For Atari Data Files

Lawrence R. Stark

*Use this "BASIC Memo Pad" program to enter a screenful of text. The computer will automatically place the screen into the pseudo string-array B$.*

Screen editing is a very convenient means for entry of data and text in computer files. Yet one of the ironies surrounding several models of small computers is that, while they have this means to edit the source code of the BASIC language programs which they all feature, the user is often reduced to some form of serial prompting for the entry of data.

There are various solutions to this problem, but one of the nicer is presented in the 400/800 Atari and its 8K BASIC. The key is the ability to control the screen and to "dump" its contents quickly with the "dynamic keyboard" technique.

The following short program is a routine which demonstrates the principle. Extracted from a larger program which externally resembles the data file manager carried in the November, 1981, issue of **COMPUTE!**, the routine presented here does little other than demonstrate a method that has been mentioned at various places in the literature on the Atari computers. The demonstration is, in effect, a "BASIC Memo Pad."

The routine is very simple. The margins of the screen are set, making a sort of sub screen. In the example, the screen dimensions are 16 lines vertically and 35 horizontally. The row of numbers that appears on the left of the screen may be said to be outside the screen. The programmer can make this left margin broader and put prompting headers in it.

Once the screen is set, characters are taken from the keyboard and displayed in the subscreen. Troublesome characters like ATASCII 125, clear screen, are bypassed. Then the routine checks to see if the border of the subscreen is crossed, PEEKing locations 84 and 85. Carriage return has no effect other than to print a carriage return that becomes, in essence, a cursor control. A "home" key is provided in

Control T. Within these confines – the loop from 150 to 220 – the user can do almost anything.

Once the data is acceptable, the loop is exited. In the demo program, Control J is used as the signal. The program prompts for a confirmation at this stage, and if a "Y" is issued, passes to the dynamic keyboard routine.

In the dynamic keyboard routine, the cursor is POSITIONed, an INPUT from the editor is requested, and the dynamic keyboard is activated and de-activated as it passes each of the 16 lines. The necessary POKEs are illustrated in lines 260 and 270. The effect is to read from the screen memory into the variable named A$.

At this point, the needs of the application program come into play. In the larger programs in which I use this routine, A$ is written to a disk file, sometimes with a second line concatenated. In the demonstration program, I have merely put it into the pseudo-string array called B$.

The demo program STOPs upon the transfer of the screen display to B$ via A$. If CONT is issued at this stage the process will come full cycle, placing B$ into display. While this is largely useless as presented here, it is similar to the method of recalling data from a disk file and displaying it on the screen. The programmer may then, once again, invoke the editing routine for updating purposes, as in the demo program. Of course, in an actual program, the revision would have to result in something other than the endless loop in the demonstration program!

In actual use many other features can be added. For instance, I have included a "parser," perhaps better called an auto-return, which searches the end of a line when the cursor passes a designated position. A subroutine then positions all characters past the right-most space on the next line. This is done using the LOCATE command. Incidentally, the LOCATE command can also be used to dump this screen, but it is much slower than the dynamic keyboard approach.

It seems a little surprising that it is possible to devise a form of screen editor using the BASIC language, but here it is. Limited in some ways, it is generally more user-friendly than the serial prompt system which tells of BASIC's origins in the days of teletypewriter terminals.

**PROGRAM. A Simple Screen Editor For Atari Data Files.**

```
0 REM * * Screen editor & dump * * * * * * *
   * * * * * * * * * * * * * *
100 DIM B$(600),A$(40)
110 ? CHR$(125)
120 GOSUB 500
130 OPEN #1,4,0,"K:"
140 REM * * * * * * * * * * * * * * * * * *
    * * * * * * * * * * * * * *
150 POSITION 3,0:? :GOSUB 1000
160 GET #1,T:IF T=10 THEN 230:REM 10 = CTRL
    "J"
170 IF T=20 THEN 150:REM 20 = CTRL "T"
180 IF PEEK(85)=36 THEN GOSUB 1000
190 IF PEEK(85)>38 THEN POKE 85,38:GOSUB 100
    0:GOTO 160
200 IF PEEK(84)>16 OR PEEK(84)<1 THEN 150
210 IF T=156 OR T=157 OR T=125 THEN 160
220 ? CHR$(T);:GOTO 160:REM main working loo
    p (160-220)
230 POSITION 3,18:? "ARE YOU SURE?";:GET #1,
    X:POSITION 3,18:? "{14 SPACES}"
240 IF X<>ASC("Y") THEN 150
250 REM * * * * * * * * * * * * * * * * * *
    * * * * * * * * * * * * * *
260 FOR I=1 TO 16:POSITION 3,I:POKE 842,13:I
    NPUT A$
270 POKE 842,12
280 B$(I*35-34,I*35)=A$
290 NEXT I
300 REM * * * * * * * * * * * * * * * * * *
    * * * * * * * * * * * * * *
310 STOP
315 ? CHR$(125)
320 FOR I=1 TO 16
330 ? B$(I*35-34,I*35)
340 NEXT I
350 GOSUB 500:GOTO 150
360 STOP
500 B$(1)=" ":B$(600)=" ":B$(2)=B$
600 POKE 82,0:POSITION 0,0:? :FOR I=1 TO 16:
    ? I:NEXT I:POKE 82,4
610 POSITION 3,0:? "|USE CTRL "J" TO ACCEPT|
    ":RETURN
1000 FOR I=1 TO 5:SOUND 0,50,10,10:NEXT I:SO
     UND 0,0,0,0:RETURN
```

# Plotting Made Easy

John Scarborough

*This utility lets you draw a figure on any graphics screen with a joystick. It will then convert it into a series of PLOT, DRAWTO statements which are automatically entered into the program. A real time-saver – you may never need graph paper again!*

If you've ever tried to PLOT and DRAWTO your way through some complex or even simple figure in one of the graphics modes, then you know how time consuming it is. So why not have the computer do it for you? This program allows you to first draw your figure on the screen using a joystick, and then have the computer do the PLOTs and DRAWTOs for you.

Enter the program into your computer carefully. Don't try to shorten it, or you're sure to get confused. I "squeezed" the program to get as much free RAM as I could. If you still need more, you can get rid of some lines from 603 on up. You don't want to go any higher than 523. Just make sure that L=650 on the last one you leave in. Before you use the program, it'll be useful to make a couple of back-up copies.

The following steps show you how to make a box with an x in it (at first glance it might look long and confusing, but once you get the hang of it you'll be able to move quickly).

1. Choose the graphics mode you want (I suggest you practice on mode 3 until you get the feel of what you're doing).
2. Choose the cursor color.
3. Choose the cursor luminance (7 will give a nice shade).
4. Choose the background color.
5. Move the cursor to a starting point and draw the box (press the joystick button to draw).
6. Position the cursor in one of the corners (corner 1), and then hit "P" (you are PLOTting your first line).
7. Move to the next corner (it doesn't matter how you get there), and hit "D." You just had the computer PLOT and DRAWTO for you.
8. For the second line: Stay in corner 2, and hit "P." Move to the third corner and hit "D."
9. Follow the same procedure until you get back to the first corner.

10. After you end your fourth line, start another one (by pressing "P"). Move to the diagonal corner (line not needed), and end the line.

11. Do the same with the other two corners.

12. Hit the ESCape key.

13. You'll briefly see some program statements being entered into the program; then you'll be in the graphics mode.

14. You should have your box with the X in it.

Lines 700-898 are saved for your PLOT-DRAWTO statements. You can LIST them at any time, but don't run it to get back to the drawing. Enter "G.4000" instead. You will find that the cursor moves very slowly in mode 8. If you can't handle the slowness then hit "F." You'll have to slow it back down (by pressing "S") when you want to PLOT. If you want to start a new figure or drawing you'll have to take out lines 700-898. Just make sure line 700 reads: 700 RETURN. A faster way to take out the lines would be to have the computer print out the line numbers for you. Then you press RETURN after each line. Example: 5000 FOR X=701 TO 721:?X:NEXT X. If your figure consumed a lot of lines it might be faster just to CLOAD your fresh program from tape.

## PROGRAM. Plotting Made Easy.

```
1 REM PLOTTING MADE EASY
2 REM BY JOHN SCARBOROUGH
3 DIM AN$(1):LI=699:GRAPHICS 0:SETCOLOR 2,0,
  0:? :? :? "GRAPHICS MODE(3 TO 8)";:INPUT M
  ODE:IF MODE=8 THEN 5
4 ? :? "CURSOR COLOR(0 TO 14)";:INPUT CC:? :
  ? "CURSOR LUMINENCE(0 TO 14)";:INPUT CL
5 ? :? "BACKGROUND COLOR(YES OR NO)";:INPUT
  AN$:IF AN$="N" THEN 7
6 ? :? "BACKGROUND COLOR(0 TO 14)";:INPUT BC
  :? :? "BACKGROUND LUMINENCE(0 TO 14)";:INP
  UT BL
7 IF MODE=8 THEN HRNG=319:VRNG=191
8 IF MODE=6 OR MODE=7 THEN HRNG=159:VRNG=95
9 IF MODE=4 OR MODE=5 THEN HRNG=79:VRNG=47
11 IF MODE=3 THEN HRNG=38:VRNG=23
14 GRAPHICS MODE:SETCOLOR 2,BC,BL:SETCOLOR 0
   ,CC,CL:SETCOLOR 4,BC,BL:COLOR 1:POKE 752,
   1:L=500:GOSUB 700
17 ? "HORIZONTAL-":? "VERTICAL-"
20 GOSUB 100:IF STRIG(0)=0 THEN PLOT H,V:GOT
   O 20
30 PLOT H,V:FOR X=1 TO 5:NEXT X:POSITION H,V
   :? #6;" ":GOTO 20
100 S=STICK(0):IF S=11 THEN H=H-1
105 IF S=5 THEN H=H+1:V=V+1
110 IF S=7 THEN H=H+1
115 IF S=6 THEN H=H+1:V=V-1
120 IF S=14 THEN V=V-1
130 IF S=13 THEN V=V+1
135 IF S=9 THEN H=H-1:V=V+1
140 IF S=10 THEN H=H-1:V=V-1
141 IF H<0 THEN H=0
142 IF H>HRNG THEN H=HRNG
143 IF V>VRNG THEN V=VRNG
150 IF V<0 THEN V=0
155 IF MM=0 THEN POKE 656,0:POKE 657,19:? "F
    REE MEMORY-";FRE(0):MM=1
160 IF PEEK(764)=56 THEN GOSUB 2000:POKE 656
    ,2:? "FAST(SLOW DOWN TO PLOT OR DRAWTO)
    ":FT=1:GOSUB 440
165 IF PEEK(764)=62 OR OK=0 THEN GOSUB 2000:
    POKE 656,2:? "  **** PLOTTING MADE EASY
    ****{3 SPACES}":FT=0:GOSUB 440:OK=1
170 IF FT=1 THEN RETURN
180 IF PEEK(764)=10 THEN GOSUB 2000:POKE 656
    ,1:POKE 657,19:? "PLOT ";H;",";V;"
    {3 SPACES}":GOSUB L:GOSUB 440
```

```
185 IF PEEK(764)=28 THEN 1010
190 IF PEEK(764)=58 THEN GOSUB 2000:POKE 656
    ,1:POKE 657,19:? "DRAWTO ";H;",";V;"
    {3 SPACES}":GOSUB L:GOSUB 440
210 POKE 656,0:POKE 657,13:? H;"   "
230 POKE 657,11:? V;"   "
300 RETURN
440 POKE 764,255:RETURN
500 L=503:A=H:B=V:RETURN
503 L=510:C=H:D=V:RETURN
510 L=513:E=H:F=V:RETURN
513 L=520:G=H:I=V:RETURN
520 L=523:J=H:K=V:RETURN
523 L=530:M=H:N=V:RETURN
530 L=533:O=H:P=V:RETURN
533 L=540:Q=H:R=V:RETURN
540 L=543:SS=H:T=V:RETURN
543 L=550:U=H:W=V:RETURN
550 L=553:Y=H:Z=V:RETURN
553 L=560:AA=H:AB=V:RETURN
560 L=563:AC=H:AD=V:RETURN
563 L=570:AE=H:AF=V:RETURN
570 L=573:AG=H:AH=V:RETURN
573 L=580:AI=H:AJ=V:RETURN
580 L=583:AK=H:AL=V:RETURN
583 L=590:AM=H:AN=V:RETURN
590 L=593:AO=H:AP=V:RETURN
593 L=600:AQ=H:AR=V:RETURN
600 L=603:AS=H:AT=V:RETURN
603 L=650:AU=H:AV=V:RETURN
650 GOTO 1010
700 RETURN
899 RETURN
950 IF DD=0 AND EE=0 AND FF=0 AND GG=0 THEN
    RETURN
951 LI=LI+1:? LI;" PL.";DD;",";EE;":DR.";FF;
    ",";GG:RETURN
1010 GRAPHICS 0:? :? :?
1015 DD=A:EE=B:FF=C:GG=D:GOSUB 950:DD=E:EE=F
     :FF=G:GG=I:GOSUB 950:DD=J:EE=K:FF=M:GG=
     N:GOSUB 950
1020 DD=O:EE=P:FF=Q:GG=R:GOSUB 950:DD=SS:EE=
     T:FF=U:GG=W:GOSUB 950:DD=Y:EE=Z:FF=AA:G
     G=AB:GOSUB 950
1030 DD=AC:EE=AD:FF=AE:GG=AF:GOSUB 950:DD=AG
     :EE=AH:FF=AI:GG=AJ:GOSUB 950:DD=AK:EE=A
     L:FF=AM:GG=AN:GOSUB 950
1040 DD=AO:EE=AP:FF=AQ:GG=AR:GOSUB 950:DD=AS
     :EE=AT:FF=AU:GG=AV:GOSUB 950:? "G.4000"
1100 POSITION 0,0:POKE 842,13:END
```

```
2000 SOUND 0,17,10,10:FOR X=1 TO 7:NEXT X:SO
     UND 0,0,0,0:RETURN
4000 A=0:B=0:C=0:D=0:E=0:F=0:G=0:I=0:J=0:K=0
     :M=0:N=0:O=0:P=0:Q=0:R=0:SS=0:T=0:U=0:W
     =0
4001 Y=0:Z=0:AA=0:AB=0:AC=0:AD=0:AE=0:AF=0:A
     G=0:AH=0:AI=0:AJ=0:AK=0:AL=0:AM=0:AN=0:
     AO=0:AP=0:POKE 842,12
4002 AQ=0:AR=0:AS=0:AT=0:AU=0:AV=0:LI=LI+1:M
     M=0:OK=0:GOTO 14
```

# Graphics Generator

Matthias M. Giwer

*Create graphics characters, SAVE them to disk, and use them in other programs.*

Recently my son has shown a distinct interest in learning to program a computer. Although I do not expect much to come of this interest at six years of age, I began working up some simple illustrative programs on programming concepts. The first was a race track for a number to go around to demonstrate a loop. The second was a Y-shaped branch for a number to go through. After the second tedious construction of the branch using line numbers, POSITION statements, and PRINT statements, it was apparent that there had to be a better way. Here is my approach to that better way.

This program permits graphics characters – or any characters – on a Graphics 0 screen. The finished screen is written to a disk file of line numbers which can then be merged with a master program by means of the ENTER command.

After running the program and giving a filename to save the results, you must not do anything to scroll the screen. This means that what you draw must be done with the cursor and you must never hit RETURN. It is recommended that the first thing you do is erase the STOPPED AT message. Do not erase or move the CONT on line 22. You may use the cursor keys and any other screen editing functions of the Atari. When finished, move the cursor down to the line containing the CONT and hit RETURN. The program will execute. When the disk drive stops, the program has finished.

There are many options available within the program. The one option you do not have is to move CONT to the last line, for if you do, the screen will scroll when you hit RETURN. Otherwise, it is rather flexible. If you wish to compose several graphics screens, run the program once for each screen. You will use a different file specification and change the value of 30000 in line 2020. Since exactly 40 lines are required to save a screen, increase 30000 in blocks of 50 so you will have free lines for RETURN statements and so forth. When putting together your finished program graphics, simply ENTER all of the file specifications you have used and LIST them under one new file specification.

The program itself simply constructs R$ to look like a line of BASIC. It concatenates (adds together) a line number (30000 + I), a print command in the form of a question mark, and then adds a quotation mark in the form of CHR$(34). (Otherwise it would be interpreted by the computer as a closing quotation mark.) Note that a quotation mark is not permitted on your graphics screen. Using the LOCATE instruction, each position of the screen is examined and added to R$ by the CHR$(X) instruction. In line 2030 J starts at 2 to coincide with the default values of the screen. If you intend to use the resulting lines with different screen widths, then this value should be changed to coincide with them. R$ is completed with a closing CHR$(34), a semicolon to prevent scrolling CHR$(59), and a carriage return CHR$ (155).

Upon ENTERing these lines you will need to do a bit of editing. First you must remove the word CONT from the next to the last line and the cursor from the last line. After this you may change blank lines to simple PRINT or ? statements. Leaving the lines in this form takes up only a few bytes and gives you what might be called relocatable graphics.

To obtain fixed location graphics, make the changes in Program 2. These will result in absolute positioning of your graphics. The POSITION statements generated by these lines will place the graphics exactly where you drew them.

**PROGRAM 1. Graphics Generator.**

```
100 GRAPHICS 0:DIM R$(80),F$(17)
120 ? "ENTER DISK NAME TO SAVE UNDER
    {10 SPACES}(Dn:filespec.ext)";:INPUT F$
140 GRAPHICS 0:POSITION 4,22:? "CONT{5 UP}":
    STOP
1900 OPEN #1,8,0,F$:J=0
2010 FOR I=0 TO 23
2020 R$=STR$(30000+I)
2026 R$(LEN(R$)+1)=" ?"
2028 R$(LEN(R$)+1)=CHR$(34)
2030 FOR J=2 TO 39
2050 LOCATE J,I,X
2060 R$(LEN(R$)+1)=CHR$(X)
2070 NEXT J
2080 R$(LEN(R$)+1)=CHR$(34)
2082 R$(LEN(R$)+1)=CHR$(59)
2090 R$(LEN(R$)+1)=CHR$(155)
2100 ? #1;R$
2110 R$=""
2120 NEXT I
2190 CLOSE #1
2200 REM
2201 GOTO 2200
```

**PROGRAM 2. Graphics Generator.**

```
2022 R$(LEN(R$)+1)="POSITION 3,"
2023 R$(LEN(R$)+1)=STR$(I)
2024 R$(LEN(R$)+1)=":"
```

# Analyze Your Program – An Atari BASIC Utility

Fred Pinho

*This program allows you to study the effects of space allocation in Atari variable value and string/array tables. You'll discover the memory saving effects of various methods of handling heavily edited programs. It's also a useful debugging tool for more advanced programmers.*

This program was inspired by Art McGraw's "Variable Name Utility" (**COMPUTE!**, Oct. 1981, #17). To do advanced programming in BASIC, one often needs information, not only on the variable name table, but also on the variable value and the string/array tables. These tables reside in memory as follows:

| | |
|---|---|
| Variable Name Table | Increasing |
| Variable Value Table | Memory |
| BASIC Program | Locations |
| String/Array Table | |

There are a set of zero-page pointers that point to these tables and enable BASIC (and the programmer) to keep track of their location.

| MEMORY LOCATION OF POINTER | MEMORY AREA POINTED TO |
|---|---|
| 130,131 | Start of Variable Name Table. |
| 132,133 | End of Variable Name Table. Points to a zero dummy byte when there are less than 128 variables. Otherwise points to the last byte of the last variable name. |
| 134,135 | Start of Variable Value Table. |
| 140,141 | Start of the String/Array Table. |
| 142,143 | Start of Run Time Stack. Also defines the end of the String/Array Table. |

As usual, these pointers point to the address in low-high format (low byte of 16 bit address is stored first). To find the complete address:

Address = PEEK (Lo Byte) + 256*PEEK (Hi Byte)

In order to be able to read the information displayed by the program, I've included a description of each of these tables.

VARIABLE NAME TABLE

Lists all the variable names in the order entered by the program. Each element of the name is stored as ATASCII characters. There are three types of variables:

1. Scalar variables – These contain a numeric value. The most significant bit is set on the last character of the name.

2. String variables – The last character stored is a $ with the most significant bit set.

3. Array variables – The last character stored is a ( with the most significant bit set.

VARIABLE VALUE TABLE

This table reserves eight bytes for each variable in the program. The first byte of each entry defines the type of variable: zero for a scalar variable, 65 for a properly-dimensioned array variable and 129 for a properly-dimensioned string variable. The second byte is the variable number (0-127). The remaining 6 bytes vary with the type of variable:

1. Scalar – The number stored in 6-byte BCD (Binary Coded Decimal) format.

2. Array – Bytes 3 and 4 give the location of the array as an offset from the beginning of the String/Array Table. Bytes 5 and 6 give the size of the first dimension of the array plus one. Bytes 7 and 8 give the size of the second array DIMension plus one. All these byte-pairs give the number in low-byte, high-byte format. To get the desired value you must again calculate by: (value in lo byte) +256* (value in hi byte).

3. String – Bytes 3 and 4 give the location of the string as an offset from the beginning of the String/Array Table. Bytes 5 and 6 give the current length of the string (i.e., the length of the string actually written to). Bytes 7 and 8 give the DIMensioned length of the string.

STRING/ARRAY TABLE

This block of memory stores all the actual string and array data. Each string character is stored as a one-byte ATASCII entry. Each element in an array is stored as a six-byte BCD number. BASIC allocates memory space within this table as dictated by the DIMension statements it encounters. As you can see, it is much more costly in memory usage to store arrays than strings. (See Program 1.) Ideally, this utility should be written with no declared variables to give a "pure" analysis of the variable tables. However, this would give a messy looking program and take a lot more coding. Therefore, I've used four variables in this program. I've chosen names that are unlikely to be used in normal programs. These are:

OPQ – FOR - NEXT counter
RST – Variable number
UVW – Location within Variable Value Table
XYZ – Location within Variable Name Table

If, for some reason, you are using these variables in your program, change the variable names in the utility program. The utility variables will be printed last and can be ignored.

A description of the program by line number is given.

| | |
|---|---|
| 18990 | Vanity line |

| | |
|---|---|
| 19010 | Opens a file to the printer. It is best to do this rather than use LPRINT since LPRINT causes formatting difficulties. |
| 19025-19040 | Prints variable names. Note that since each name is ended with a character whose most significant bit is set (i.e., an inverse character), this bit is stripped out before printing. This would not be necessary if printing to the screen. |
| 19050-19070 | Checks for type of variable by inspecting first bit of entry in variable value table. Directs program to proper subroutine |
| 19080 | Checks for error in value table |
| 19090 | Increments variables |
| 19100 | Checks for end of variable name table |
| 19110 | Prints number of variables found which equals the total variables minus the four in the utility program. |
| 19120 | Prints memory size of string/array table. |
| 19130 | Closes printer file |
| 19140 | Error routine. Prints number found |
| 19200-19240 | Scalar variable subroutine. Converts six-byte BCD number to a decimal number multiplied by a power of 100 |
| 19300-19350 | Array variable subroutine. Calculates location of array as an offset from the start of the string/array table. Also gives the first and second DIMensions of the array. Note that the Atari stores the chosen DIMension plus one. Therefore, the program subtracts one before printing. However, the actual DIMension is one higher than the chosen and printed value since the computer starts counting from zero rather than one. |
| 19400-19450 | String variable subroutine. Calculates the string storage location as an offset from the start of the string/ array table. Also gives the DIMensioned length and the current length of the string. Note that the current length of the string is just the last location written to; there need not be anything in the previous locations. To show this, load the utility and type: 100 DIM A$ (120), B$ (120) 110 A$ (103,103) = "F":B$ = "FFFF" Then RUN the utility program (i.e., let it analyze itself). See Figure 1 for the output of this run. Also, note that string position numbering starts from one, not zero, as in arrays. |

The abbreviations used in the printout are:

| | |
|---|---|
| VAR. NO. | – Variable Number |
| VAR. NAME | – Variable Name |
| OFS | – Offset |
| DIM1 | – First DIMension of Array |
| DIM2 | – Second DIMension of Array (zero if single-dimension array) |
| CUR LTH | – Current Length of String |
| DIM LTH | – DIMensioned Length of String |

Note that the *location* of the start of each table for the analyzed program should not be found using this utility. This is because the presence of the utility program will cause a shift of the string/array table location. If you need to know these locations, PEEK the appropriate locations *before* you load the utility program.

The utility program has been written with high line numbers so that it won't interfere with most programs to be analyzed. To use this utility, type it in and then save it to either disc or cassette using the LIST command. Don't use SAVE or CSAVE, as this will prevent you from merging the utility with the program to be analyzed (the program in the computer will be wiped out as you LOAD or CLOAD the utility program). Now LOAD in your program to be analyzed. To merge the two programs, LOAD in the utility using the ENTER command. Now turn on the printer, type in GOTO 19000 (RETURN).

If you did the above, you will get some unexpected results. All the variables will be listed, but they will have no entries for them. For example, all scalar variables will be zero regardless of their value in the program. Also, all strings will be unDIMensioned and will have zero for their length. Apparently, when a program is SAVEd to disc, the Atari saves the Variable Value Table with entries set to "zero" condition. Therefore, to get the proper analysis, do the following:

1. LOAD your program.
2. RUN it!
3. ENTER the utility program.
4. Type GOTO 19000 (RETURN).

This will give you a proper analysis. Note that loop variables will not always be caught at their initial value.

If your program has line numbers in the 18990-19450 area, it could interfere with the utility. Therefore, RUN your program and then delete the problem lines. This will not affect the Variable Name and Variable Value tables. Then ENTER the utility and proceed as before.

What can you use this program for? Well, first you can use it to gain a better understanding of how BASIC works. For example, analyze a heavily edited program which has had variables deleted. If you've only SAVEd this program, you'll find that these variables are still listed in the tables and continue to consume memory. LIST your program to disc or cassette and then ENTER it back into the computer. If you now re-analyze the program, you'll find that these "phantom" variables will have been eliminated. If you check "free" memory [FRE(0)] before and after, you'll find a gain in useable memory.

Many advanced programming techniques use string manipulations to take advantage of the high speed, string handling routines in the Atari. These often depend on changing the entries in the variable value table to relocate strings under program control. This utility is useful as a debugging tool for these applications.

One final note for those who do not have printers. If you wish to output to the screen, change line 19010 to:

OPEN #3,8,0,"S:"

You can stop the screen output at any time and then resume it by "Control-1."

**Figure 1.**

```
VAR. NO. =0 VAR. NAME=A$
STRING DIMed
OFS=0:CUR LTH=103:DIM LTH=120
VAR. NO. =1 VAR. NAME=B$
STRING DIMed
OFS=120:CUR LTH=4:DIMLTH=120
VAR. NO. =2 VAR. NAME=XYZ
SCALAR—76.82000000*100
VAR. NO. =3 VAR. NAME=UVW
SCALAR—77.17000000*100
VAR. NO. =4 VAR. NAME=RST
SCALAR—04.00000000*0
VAR. NO. =5 VAR. NAME=OPQ
SCALAR—08.00000000*0
END OF VARIABLE NAME AND VALUE TABLES.
NUMBER OF VARIABLES FOUND=2
STRING/ARRAY AREA IS CURRENTLY 240 BYTES LONG.
```

## PROGRAM. Analyze Your Program – An Atari BASIC Utility.

```
18990 REM VNT/VVT UTILITY BY F. PINHO 12/22/
      81
19000 XYZ=PEEK(130)+256*PEEK(131):UVW=PEEK(1
      34)+256*PEEK(135):RST=0
19010 OPEN #3,8,0,"P:"
19020 ? #3;"VAR. NO.=";RST;"   ";:? #3;"VAR.
      NAME=";
19025 IF PEEK(XYZ)<128 THEN ? #3;CHR$(PEEK(X
      YZ));
19030 IF PEEK(XYZ)>127 THEN ? #3;CHR$(PEEK(X
      YZ)-128);
19040 IF PEEK(XYZ)<128 THEN XYZ=XYZ+1:GOTO 1
      9025
19050 IF PEEK(UVW)=0 THEN GOSUB 19200
19060 IF PEEK(UVW)=64 OR PEEK(UVW)=65 THEN G
      OSUB 19300
19070 IF PEEK(UVW)=128 OR PEEK(UVW)=129 THEN
       GOSUB 19400
19080 IF PEEK(UVW)<>0 AND PEEK(UVW)<>64 AND
      PEEK(UVW)<>65 AND PEEK(UVW)<>128 AND P
      EEK(UVW)<>129 THEN GOTO 19140
19090 UVW=UVW+8:XYZ=XYZ+1:RST=RST+1
19100 IF XYZ<(PEEK(132)+256*PEEK(133)) THEN
      19020
19110 ? #3;"END OF VARIABLE NAME AND VALUE T
      ABLES.":? #3;"NUMBER OF VARIABLES FOUN
      D=";RST-4
19120 ? #3;"STRING/ARRAY AREA IS CURRENTLY "
      ;((PEEK(142)+256*PEEK(143))-(PEEK(140)
      +256*PEEK(141)));" BYTES LONG."
19130 CLOSE #3:END
19140 ? #3:? #3;"ERROR! VARIABLE TYPE NUMBER
      =";PEEK(UVW):END
19200 ? #3:? #3;"SCALAR--";:IF PEEK(UVW+2)=0
       THEN ? #3;"ZERO":? #3:RETURN
19210 ? #3;INT(PEEK(UVW+3)/16);(PEEK(UVW+3)-
      (INT(PEEK(UVW+3)/16))*16);".";
19220 FOR OPQ=4 TO 7:? #3;INT(PEEK(UVW+OPQ)/
      16);(PEEK(UVW+OPQ)-(INT(PEEK(UVW+OPQ)/
      16))*16);
19230 NEXT OPQ
19240 ? #3;"*";:? #3;((PEEK(UVW+2)-64)*100):
      ? #3:RETURN
19300 ? #3:? #3;"ARRAY  ";
19310 IF PEEK(UVW)=64 THEN ? #3;"unDIMed";:?
       #3
19320 IF PEEK(UVW)=65 THEN ? #3;"DIMed";:? #3
```

```
19330 ? #3;"OFS=";(PEEK(UVW+2)+256*PEEK(UVW+
      3));":";
19340 ? #3;"DIM1=";((PEEK(UVW+4)+256*PEEK(UV
      W+5))-1);":";
19350 ? #3;"DIM2=";((PEEK(UVW+6)+256*PEEK(UV
      W+7))-1):? #3:RETURN
19400 ? #3:? #3;"STRING  ";
19410 IF PEEK(UVW)=128 THEN ? #3;"unDIMed";:
      ? #3
19420 IF PEEK(UVW)=129 THEN ? #3;"DIMed";:?
      #3
19430 ? #3;"OFS=";(PEEK(UVW+2)+256*PEEK(UVW+
      3));":";
19440 ? #3;"CUR LTH=";(PEEK(UVW+4)+256*PEEK(
      UVW+5));":";
19450 ? #3;"DIM LTH=";(PEEK(UVW+6)+256*PEEK(
      UVW+7)):? #3:RETURN
```

# Inside Atari Microsoft BASIC: A First Look

Jim Butterfield

*Atari's long-awaited Microsoft BASIC is here at last. Jim Butterfield, an expert on the 8K Microsoft BASIC used on other machines, begins the documentation of the complex inner workings of Atari Microsoft BASIC.*

It's a big BASIC. It occupies 18K of RAM, which means there's a lot of code in the interpreter. It also does some new things. Single versus double precision arithmetic, for example, calls for a dramatic rearrangement of the floating accumulators and of the way variables are stored as compared to the better-known 8K Microsoft BASICs.

With the expanded features come new techniques to be mastered. I wince when PRINT 10/4 yields an answer of 2 (to get 2.5, you must force floating point with PRINT 10/4.0).

## The Architecture

The following discussion assumes that users have had some exposure to the mechanics of other Microsoft BASICs.

Your BASIC program will be stored right behind the interpreter (Hex 6980 and up). It's the usual format: two-byte forward chain to the next BASIC line, two-byte BASIC line number, the line itself (tokenized) and finally a zero byte to flag end-of-line. The end-of-program is identified by zero-bytes in the forward chain location.

Variables come behind your program – check the address in hex 84 and 85, or PRINT PEEK(132) + PEEK(133) * 256 – but storage is fairly complex. The first two bytes are the first two characters of your variable name, but with many bits stripped away and replaced with "variable type" bits: don't be surprised if your variable A ends up with the name stored as value 1 rather than the ASCII 65 which corresponds to A. The third byte is the length of this variable entry. Now we have a messy bit: if you have a long variable name such as PLUGH the extra letters (UGH) are stored starting at the fourth byte. Finally, the value itself.

The following memory map is a brief list of the locations I have spotted while looking around. It's far from complete; but those who would like to rummage around will find it handy.

| | | |
|---|---|---|
| 0080-0081 | 128-129 | Pointer: Start-of-Basic |
| 0082-0083 | 130-131 | Pointer |
| 0084-0085 | 132-133 | Pointer: Start-of-Variables |
| 0086-0087 | 134-135 | Pointer: Start-of-Arrays |
| 008A-008B | 138-139 | Pointer: String storage (moving down) |
| 0094-0095 | 148-149 | Current data pointer |
| 00AE-00C5 | 174-197 | CHRGET subroutine |
| 00B4 | 180 | CHRGOT entry point |
| 00B5-00B6 | 181-182 | Basic pointer within subroutine |
| 00C7-00C8 | 199-200 | Variable pointer for FOR/NEXT |
| 00CB | 202 | $98 = READ, $40 = GET, 0 = INPUT |
| 00CD | 204 | Default DIM flag |
| 00D1 | 209 | Accum sign compare, #1 vs #2 |
| 00D2 | 210 | Accum#1 Low order (rounding) |
| 00D4 | 212 | Variable name length |
| 00D5-00D8 | 213-216 | Utility Pointer area |
| 00DC-00E4 | 220-228 | Misc numeric work area |
| 00E5 | 229 | Accum#1 precision flag |
| 00E6 | 230 | Accum#1: Sign |
| 00E7 | 231 | Accum#1: Exponent |
| 00E8-00EE | 232-238 | Accum#1: Mantissa |
| 00F0 | 240 | Accum#2: Sign |
| 00F1 | 241 | Accum#2: Exponent |
| 00F2-00F8 | 242-248 | Accum#2: Mantissa |
| 00F9-00FF | 249-255 | Product area for multiplication |
| 0480- | 1152- | Variable name setup area |
| 1F00-697F | 7936-27007 | Microsoft Basic Interpreter |
| 6980- | 27008- | Basic program staging area |

CHAPTER THREE

# ADVANCED GRAPHICS AND GAME UTILITIES

# Player-Missile Drawing Editor

E. H. Foerster

*You can toss out your graph paper and your binary to decimal conversion tables. The P/M Drawing Editor lets you design players and missiles on-screen with a joystick. Because of automatic program adjustments, you can easily visualize players of any size, including double players or combining two players into one. When you're done, you can view the player "in action," and even automatically generate a BASIC routine for using the players and missiles in your own programs. It will run in 16K.*

Would you like to write a Player-Missile (P/M) program, but are intimidated by the need to convert your drawn player on graph paper to numerical image data? With the program in this article, you can draw your P/M object using the joystick and then let your computer do the work of converting the image to numerical data for your favorite P/M movement routine.

This program will actually perform the task of writing the DATA statements containing the P/M image using a program-writing program. The complete capabilities along with explanations are included in the instructions for the program. Only limited instructions and no REMark statements are included in the program to permit its use in a 16K machine. For those interested in the details, a program outline along with a listing of variables is included.

## Using The Editor

Let's walk through a simple example of the use of the Editor. Set up a player with double line resolution, size one, and eight lines long. Place the cursor at the top right corner of the drawing easel, press "D" for draw-to, move the cursor to bottom left and push the trigger button. Bingo, you have drawn a player consisting of a diagonal line. Press "S" for stop and, when the menu is displayed, press "L" for list.

The computer now displays your player as numerical data. Notice that each number represents the bit value of the P/M pixel as you move down the player, line-by-line. To see your player in P/M graphics mode: press "V" for view and, in a few seconds, your player appears in P/M graphics.

Would you like to see your object in single line resolution and at size four? Follow the instructions in the text window and, with a few keystrokes, the changes have been accomplished. Suppose you want to make changes to the image. Just a few keystrokes and you are back to the drawing easel. However, now the easel is different from the original, in size and shape, reflecting the changes made. The program allows complete freedom when going from one area to another. This allows you to make as many changes as needed before you record your final image data on tape. You can add or delete lines at the top or bottom and a fill routine will fill in an area.

Experiment by combining up to four players, changing the parameters of size and resolution, and you will soon have a better understanding of the meaning and interrelations of these parameters.

One unique tool used in the program is the SGN function and logical operators for converting the STICK(0) readings to X,Y coordinates. The actual examples in the program may be a little obscure. The routine is as follows:

```
10 S=STICK(0)
20 IF S<12 THEN X=X+SGN(8-S):X=X+(X<XMIN)-
   (X>XMAX): S=S+(X<8)*4+4
30 IF S<15 THEN Y=Y+SGN(13.5-S):Y=Y+(Y<YMIN)-
   (Y>YMAX)
```

You may not believe it, but these three lines read all nine joystick positions, convert the reading to new X,Y coordinates, and adjust these coordinates to the limits expressed by the X and Y MAX and MIN values. For those not familiar with logical statements, the value in parentheses of a comparison or equality evaluates to a one if true and zero if not true. Such statements can frequently reduce two lines of code using IF statements to a single line of code.

A note of caution if you are planning on using the "Player Missile Graphics Made Easy," by Sak and Meier in **COMPUTE!**, February 1981, #21. The program is designed for use with single line resolution. Addresses for players in P/M memory are indexed by page number. However, in double line resolution, memory for players two and four starts at half-page intervals and cannot be accessed. The P/M drawing Editor program gets around this limitation by extending the lengths of players one and three to include players two and four respectively. This, however, works only when adjacent players are moved together.

The LPRINT in line 1580 in the program writing subroutine serves a similar purpose to the use of LPRINT before a CSAVE.

Before this statement was included, only one player could be recorded on tape. Subsequent players would give error messages when they were later entered.

## Player-Missile Drawing Editor Instructions

### A. INITIAL QUESTIONS

1. **INPUT Resolution:** See table for explanation. Resolution for all players and missiles must be the same for any one program.
2. **INPUT Player or Missile:** A player is eight P/M pixels (bits) wide; a missile is two P/M pixels wide.
3. **INPUT number of players:** There are four players available, each eight P/M pixels wide. Any number of players may be placed side by side and moved together. Thus, four players combined will give 32 P/M pixels across.
4. **INPUT Size:** See table for explanation. This parameter can be changed later in the program.
5. **INPUT** number of vertical lines. See table for explanation. Lines can be added or deleted later in program.

### B. DRAWING EASEL

The drawing easel will appear as a green area. The drawn player appears as an orange area. The cursor is indicated by a lighter colored green or orange area. To change color, press joystick trigger. To move cursor, use joystick.

**Keyboard Options**

**(0-9)** Controls speed of cursor movement. 0 = fast, 9 = slow, 2 = initial speed.

**(D)raw to:** Draw line from current cursor position. Move cursor to new position. Press trigger button to draw line between two positions.

**(F)ill:** Used to fill an area. The area must be bounded by orange on all four sides. Place cursor below highest green space and, if possible, above lowest green space. Press F. For odd-shaped areas the routine may have to be repeated.

**(L)ines:** Add or delete vertical lines at top or bottom of drawing easel.

**(S)top:** Stop drawing. If the drawing is large, there may be a considerable wait while the diagram is converted to numerical data.

### C. OPTION MENU

**(V)iew:** observe player(s) in P/M graphics. Move player using joystick. During View you may press:

**(S)ize** to change size of player,
**(C)olor** to change color of player. Horizontal movement of joystick changes color. Vertical movement of joystick changes intensity. Color number and intensity are displayed as changes are made.
**(R)esolution** to change resolution.

**(L)ist:** Get listing of image data for player(s).

**(E)dit:** Return to drawing easel. If size or resolution was changed during view, then drawing easel is modified accordingly. If size limitations are exceeded, then size is reduced to two.

**(S)ave:** Save image numerical data on tape. Insert blank tape in recorder. Press RECORD and PLAY. Answer question for player number. If this is the first player, then answer zero. If previous players were saved, enter next player number. For example: if first drawing was two players wide, they were player zero and one. You, therefore, enter two for this player number. Data will be transferred to tape using print statements. The recorder will start and stop during this procedure.

**(D)raw new player:** Used to erase current player and start new player.

## D. USE OF RECORDED DATA

Recorded data will be entered into the computer using the ENTER command. This data may be merged with a resident program. DATA statements will be written starting at line 31,000.

A. If the program is not going to be merged with a resident program, then type NEW and press RETURN.

B. Place tape in recorder.

C. Press PLAY.

For each player or group of players:

D. Type ENTER"C" and press RETURN twice.

E. READY will appear twice on screen before playback is complete.

F. Now LIST your entry.

**Table 1.**
TABLE OF GRAPHICS POINT SIZE FOR PM GRAPHICS

| Resolution | Size | P/M Pixel Vertical | Size* Horizontal |
|---|---|---|---|
| Double Line (1) | 1 | 2 | 2 |
| ” ” | 2 | 2 | 4 |
| ” ” | 4 | 2 | 8 |
| Single Line (2) | 1 | 1 | 2 |
| ” ” | 2 | 1 | 4 |
| ” ” | 4 | 1 | 8 |

*Measured in Graphics Mode 8 pixel size (the text mode cursor is eight pixels high and eight pixels wide).

**Table 2. Program Outline.**

| | |
|---|---|
| 10-20 | Initializes and defines constants. |
| 30-120 | Inputs parameters and draws initial easel. |
| 130-160 | Drawing loop. |
| 170-190 | Checks RE * NP * SI limit. |
| 200-230 | Calculates parameters for easel and draws easel. |
| 250-290 | Moves cursor. |
| 290 | Changes cursor to normal green or orange. |
| 300-350 | Loops for keyboard input during draw. |
| 400-420 | Gets keyboard entry. |
| 450-560 | Draw-to routine. |
| 600-670 | Fill routine. |
| 700-730 | Stops. |
| 750-870 | Adds or deletes lines. |
| 900-980 | Menu. |
| 1000-1210 | Views in P/M graphics. |
| 1220 | POKEs X position and size. |
| 1230 | Stops P/M View. |
| 1240 | POKEs Color. |
| 1250-1280 | Changes size. |
| 1300-1390 | Changes color. |
| 1400-1450 | Edits. |
| 1450-1480 | LISTs player data. |
| 1500-1610 | Programs writing program for tape. |

**Table 3. Constants Used In Program.**

| | |
|---|---|
| **P/M** = | P/M page. |
| **P/MB** = | P/M base address. |
| **P(A)** = | Player A image data address. |
| **PY(A)** = | Player A Y-position in player area. |
| **PX(A)** = | Player A X-position. |
| **PA** = | Pause for adjusting speed of cursor in drawing program. |
| **LA** = | Line advance subroutine in program writing program. |
| **CO** = | Color: 16*Color no. + intensity. |
| **PLX** = | Player 0 horizontal position register. |
| **PLY** = | Player 0 vertical position register. |
| **PLL** = | Player length register. |
| **RE** = | Resolution. |
| **MI** = | Missile flag. |
| **NP** = | No. of players |
| **SI** = | Horizontal P/M size. |
| **VL** = | No. of vertical lines for player. |
| **G** = | Graphics determinant. |
| **XD** = | X draw-to dimension for each P/M pixel. |
| **H** = | Horizontal determinant. |
| **X1, Y1** = | Temporary X and Y coordinates. |
| S = | STICK(0) |

## PROGRAM. Player-Missile Drawing Editor.

```
10 DIM A$(1),P(3),PY(3),PX(3):PA=2:LA=1590:C
   O=24:PLX=53248:PLY=1780:PLL=1784
15 IF PEEK(1536)<>162 THEN GOSUB 9000
20 PM=PEEK(106)-16:PMB=PM*256:POKE 54279,PM:
   FOR A=0 TO 3:P(A)=PMB+A*24:NEXT A
30 GRAPHICS 0:? :? "INPUT VERTICAL RESOLUTIO
   N:":? "  (1)=DOUBLE LINE RESOLUTION":? "
    (2)=SINGLE LINE RESOLUTION"
40 TRAP 30:INPUT RE:IF RE>2 OR RE<1 THEN 30
50 ? :? "DO YOU WANT TO DRAW (P)LAYER OR
   {7 SPACES}(M)ISSILE";:INPUT A$:IF A$<>"P"
    AND A$<>"M" THEN 50
60 IF A$="M" THEN MI=1:NP=1:GOTO 80
70 ? :? "HOW MANY PLAYERS DO YOU WANT TO COM
   BINE TO FORM A PLAYER(1-4)":TRAP 70:INPUT
    NP:IF NP<1 OR NP>4 THEN 70
80 NP=NP-1:? :? "INPUT HORIZONTAL SIZE (1,2
   OR 4)":TRAP 80:INPUT SI:IF SI<1 OR SI>4 O
   R SI=3 THEN 80
90 GOSUB 170
100 ? :? "HOW MANY LINES DO YOU WANT FOR YOU
    R{3 SPACES}PLAYER(1-24)":TRAP 100:INPUT
    VL:IF VL<1 OR VL>24 THEN 100
110 GOSUB 200:SETCOLOR 2,12,6:COLOR 3:X=20*G
    :Y=10*G-1:PLOT X,Y:DRAWTO X+XD,Y
120 IF STRIG(0)=0 THEN POKE 710,42+(PEEK(710
    )=42)*156:PLOT X,Y:DRAWTO X+XD,Y
130 S=STICK(0):IF S<15 THEN GOSUB 250
140 IF PEEK(53775)<255 THEN 300
150 FOR A=1 TO 25*PA:NEXT A
160 GOTO 120
170 IF RE*(NP+1)*SI<17 THEN RETURN
180 GRAPHICS 0:? :? "THIS PROGRAM CANNOT HAN
    DLE 4 PLAYERS, SINGLE LINE RESOLUTION AN
    D SIZE"
190 ? "GREATER THAN 2.  THE SIZE WILL BE
    {5 SPACES}CHANGED TO 2.":SI=2:? :? "PRES
    S RETURN TO CONTINUE":INPUT A$:RETURN
200 XD1=RE*SI:XD=XD1-1:H=(NP+1)*XD1:IF MI TH
    EN H=H/4
210 G=1+((H>5) OR (VL>20))+2*((H>10) OR (VL>
    40)):GRAPHICS 19+(G>1)*G
220 YMIN=INT(10*G-VL/2):YMAX=INT(10*G+VL/2-1
    ):XMIN=20*G-4*H:XMAX=20*G+4*H-1-XD
230 SETCOLOR 0,2,8:SETCOLOR 1,12,4:COLOR 2:F
    OR Y=YMIN TO YMAX:PLOT XMIN,Y:DRAWTO XMA
    X+XD,Y:NEXT Y:RETURN
250 GOSUB 290
```

```
260 IF S<12 THEN X=X+SGN(8-S)*XD1:X=X+(X<XMI
    N)*XD1-(X>XMAX)*XD1:S=S+(S<8)*4+4
270 IF S<15 THEN Y=Y+SGN(13.5-S):Y=Y+(Y<YMIN
    )-(Y>YMAX)
280 LOCATE X,Y,A:POKE 710,PEEK(707+A)+2:COLO
    R 3:PLOT X,Y:DRAWTO X+XD,Y:RETURN
290 COLOR 1+(PEEK(710)=198):PLOT X,Y:DRAWTO
    X+XD,Y:RETURN
300 GOSUB 400:IF A=ASC("D") THEN 450
310 IF A=ASC("F") THEN 600
320 IF A=ASC("S") THEN GOSUB 700:GOTO 900
330 IF A=ASC("L") THEN GOSUB 700:GOTO 750
340 IF A<58 AND A>47 THEN PA=A-47
350 GOTO 120
400 OPEN #1,4,0,"K:":GET #1,A:CLOSE #1
410 IF PEEK(53775)<255 THEN 410
420 RETURN
450 X1=X:Y1=Y
460 S=STICK(0):IF S=15 THEN 460
470 GOSUB 260
480 IF STRIG(0)=0 THEN 510
490 S=STICK(0):IF S<15 THEN GOSUB 250
500 GOTO 480
510 COLOR 1:X2=X:Y2=Y:Y3=Y1:A=XD1*SGN(X2-X1)
    :B=SGN(Y2-Y1):C=X2-X1+A:D=Y2-Y1+B
520 IF C=0 OR D=0 THEN FOR A=0 TO XD:PLOT X1
    +A,Y1:DRAWTO X2+A,Y2:NEXT A:GOTO 570
530 FOR X=X1 TO X2 STEP A:FOR Y=Y3 TO Y2 STE
    P B:PLOT X,Y:DRAWTO X+XD,Y
540 IF (X-X1+A)/C=(Y-Y1+B)/D THEN Y=Y+B:GOTO
     560
550 IF (X-X1+A)/C>(Y-Y1+B)/D THEN NEXT Y
560 Y3=Y:NEXT X:X=X-A:Y=Y-B
570 GOSUB 280:GOTO 120
600 X1=X:COLOR 2:PLOT X,Y:DRAWTO X+XD,Y:B=67
    0
610 GOSUB B:LOCATE X,Y-1,A:IF A=2 THEN Y=Y-1
    :GOTO 610
620 GOSUB B:LOCATE X-1,Y,A:IF A=2 THEN X=X-1
    :GOTO 620
630 GOSUB B:COLOR 1:PLOT X,Y
640 GOSUB B:LOCATE X+1,Y,A:IF A=2 THEN X=X+1
    :PLOT X,Y:GOTO 640
650 X=X1:Y=Y+1:GOSUB B:LOCATE X,Y,A:IF A=2 T
    HEN 620
660 GOSUB 280:GOTO 130
670 IF X=XMAX OR X=XMIN OR Y=YMAX OR Y=YMIN
    THEN POP :GOSUB 280:GOTO 120
680 RETURN
700 GOSUB 290:FOR A=0 TO NP:B=0:X1=XMIN+8*A*
```

```
    XD1:FOR Y=YMIN TO YMIN+VL-1
710 C=128:D=0:FOR X=X1 TO X1+7*XD1 STEP XD1:
    IF MI THEN FOR X=XMIN TO XMIN+XD1 STEP X
    D1
720 LOCATE X,Y,E:IF E=1 THEN D=D+C
730 C=C/2:NEXT X:B=B+1:POKE P(A)+B,D:NEXT Y:
    NEXT A:RETURN
750 D=0:T=0:? :? "(A)DD OR (D)ELETE LINES":I
    NPUT A$:IF A$="D" THEN D=1:GOTO 770
760 IF A$<>"A" THEN 750
770 ? :? "AT (T)OP OR (B)OTTOM":INPUT A$:IF
    A$="T" THEN T=1:GOTO 790
780 IF A$<>"B" THEN 770
790 ? :? "HOW MANY LINES":TRAP 790:INPUT C:I
    F D THEN C=C-(C-VL+1)*(C>=VL):GOTO 840
800 IF VL+C>24 THEN C=24-VL:? :? "EXCEEDED L
    IMIT":? :? "WILL ADD ONLY ";C;" LINES":?
     :? "PRESS RETURN":INPUT A$
810 IF T THEN FOR A=0 TO NP:FOR B=VL TO 0 ST
    EP -1:POKE P(A)+B+C,PEEK(P(A)+B):NEXT B:
    NEXT A:GOSUB 860
820 VL=VL+C:IF  NOT D AND  NOT T THEN GOSUB
    870
830 GOTO 1400
840 IF T THEN FOR A=0 TO NP:FOR B=C TO VL:PO
    KE P(A)+B-C,PEEK(P(A)+B):NEXT B:NEXT A
850 FOR A=0 TO NP:FOR B=VL-C+1 TO VL:POKE P(
    A)+B,0:NEXT B:NEXT A:VL=VL-C:GOTO 1400
860 FOR A=0 TO NP:FOR B=1 TO C:POKE P(A)+B,0
    :NEXT B:NEXT A:RETURN
870 FOR A=0 TO NP:FOR B=VL-C+1 TO VL:POKE P(
    A)+B,0:NEXT B:NEXT A:RETURN
900 GRAPHICS 0:? :? "DO YOU WANT TO:":? :? "
      (V)IEW PLAYER IN PM GRAPHICS?":? "  (R
    )ECORD DATA?":? "  (L)IST DATA?"
910 ? "  (E)DIT PLAYER?":? "  (B)EGIN WITH A
     NEW PLAYER?":? "  (S)TOP?"
920 GOSUB 400:IF A=ASC("V") THEN 1000
930 IF A=ASC("R") THEN 1500
940 IF A=ASC("L") THEN 1450
950 IF A=ASC("E") THEN GOSUB 170:GOTO 1400
960 IF A=ASC("B") THEN RUN
970 IF A=ASC("S") THEN STOP
980 GOTO 900
1000 TRAP 40000:GOSUB 1240
1010 GRAPHICS 7:GRAPHICS 3:POKE 559,46+16*(R
     E=2):POKE 53277,3:GOSUB 1220:Y=64*RE-VL
     /2
1020 FOR A=1774 TO 1787:POKE A,0:NEXT A
1030 IF RE=1 AND NP>0 THEN FOR A=0 TO 1:POKE
```

```
      PLL+A,VL+128:NEXT A:GOTO 1050
1040 FOR A=0 TO NP:POKE PLL+A,VL:NEXT A
1050 PXM=255-NP*8*SI
1060 FOR A=0 TO 3:PY(A)=PMB+512*RE+128*RE*A:
     NEXT A
1070 FOR A=0 TO NP:FOR B=1 TO VL:POKE PY(A)+
     B,PEEK(P(A)+B):NEXT B:NEXT A
1080 POKE 1788,PM+2*RE:Z=USR(1696)
1090 ? :? "'RETURN' TO MAIN PROGRAM":? "'C'
     TO CHANGE COLOR":? "'S' TO CHANGE SIZE"
1100 ? "'R' TO CHANGE RESOLUTION FROM ";RE;"
      TO ";3-RE;
1110 S=STICK(0):X=0
1120 IF S<12 THEN X=SGN(8-S):S=S+(S<8)*4+4:X
     =X+(PX(0)+X<0)-(PX(0)+X>PXM)
1130 FOR A=0 TO NP:PX(A)=PX(A)+X:NEXT A
1140 IF S<15 THEN Y=Y+SGN(13.5-S):Y=Y+(Y<0)-
     (Y>255-VL)
1150 FOR A=0 TO NP:POKE PLY+A,Y:POKE PLX+A,P
     X(A):NEXT A
1160 IF PEEK(53775)=255 THEN 1110
1170 GOSUB 400:IF A=155 THEN GOSUB 1230:GOSU
     B 170:GOTO 900
1180 IF A=ASC("S") THEN 1250
1190 IF A=ASC("C") THEN 1300
1200 IF A=ASC("R") THEN RE=3-RE:GOSUB 1230:G
     OTO 1010
1210 GOTO 1110
1220 PX(0)=128-4*(NP+1)*SI:FOR A=0 TO NP:PX(
     A)=PX(0)+8*SI*A:POKE PLX+A,PX(A):POKE 5
     3256+A,SI-1:NEXT A:RETURN
1230 FOR A=0 TO 3:POKE PLX+A,0:POKE PLY+A,0:
     NEXT A:RETURN
1240 FOR A=704 TO 707:POKE A,CO:NEXT A:RETUR
     N
1250 ? "SIZE=";SI:? "PRESS 1,2 OR 4 TO CHANG
     E SIZE":? "PRESS 'RETURN' TO GET OUT OF
      SIZE{5 SPACES}SUBROUTINE";
1260 GOSUB 400:IF A=155 THEN 1090
1270 IF A=49 OR A=50 OR A=52 THEN SI=A-48:GO
     SUB 1220
1280 GOTO 1250
1300 A=INT(CO/16):B=CO-A*16:? :? "COLOR=";A,
     "INTENSITY=";B
1310 ? "MOVE JOYSTICK HORIZONTALLY TO CHANGE
       COLOR; VERTICALLY TO CHANGE INTENSITY
     "
1320 ? "PRESS RETURN TO EXIT SUBROUTINE";
1330 S=STICK(0):IF PEEK(53775)<255 THEN 1380
1340 IF S=15 THEN 1330
```

```
1350 IF S=7 OR S=11 THEN A=A+SGN(8-S):A=A-(A
     >15)*16+(A<0)*16
1360 IF S=13 OR S=14 THEN B=B-2*SGN(13.5-S):
     B=B-(B>14)*16+(B<0)*16
1370 CO=16*A+B:GOSUB 1240:GOTO 1300
1380 GOSUB 400:IF A=155 THEN 1090
1390 GOTO 1300
1400 GOSUB 200:COLOR 1:FOR A=0 TO NP:X=XMIN+
     A*XD1*8:FOR Y=1 TO VL
1410 B=PEEK(P(A)+Y):C=128:FOR X1=0 TO 7:IF M
     I THEN FOR X1=0 TO 1
1420 IF B>=C THEN B=B-C:GOSUB 1440
1430 C=C/2:NEXT X1:NEXT Y:NEXT A:X=20*G:Y=10
     *G-1:GOSUB 280:GOTO 120
1440 PLOT X+X1*XD1,YMIN+Y-1:DRAWTO X+X1*XD1+
     XD,YMIN+Y-1:RETURN
1450 GRAPHICS 0:FOR A=0 TO NP:? :IF MI THEN
     ? "MISSILE":GOTO 1470
1460 ? "PLAYER ";A
1470 FOR B=1 TO VL:? PEEK(P(A)+B);",";:NEXT
     B:? "{BACK S}":NEXT A
1480 ? :? "PRESS RETURN TO RETURN TO PROGRAM
     ":GOSUB 400:GOTO 900
1500 ? :? "WHAT NO. IS THIS PLAYER,MISSILE O
     R 1STPLAYER OF GROUP":TRAP 900:INPUT PN
1510 ? :? "PLACE BLANK TAPE IN RECORDER, PRE
     SS PLAY,RECORD AND RETURN"
1520 OPEN #1,8,0,"C:":FOR A=1 TO 64:? #1;"R.
     ";:NEXT A
1530 FOR A=0 TO NP:LI=31400+(PN+A)*100:? #1:
     IF MI THEN 1600
1540 ? #1;LI;"REM** PLAYER ";PN;":RESOLUTION
     ,PLAYER LENGTH,SIZE,COLOR. SUBSEQUENT L
     INE IS IMAGE DATA":GOSUB LA
1550 ? #1;LI;"DATA ";RE;",";VL;",";SI;",";CO
     :GOSUB LA
1560 FOR B=1 TO VL:IF B=1 OR B=26 THEN ? #1:
     ? #1;LI;"DATA ";:GOSUB LA:GOTO 1580
1570 ? #1;",";
1580 ? #1;PEEK(P(A)+B);:NEXT B:NEXT A:CLOSE
     #1:TRAP 900:LPRINT :GOTO 900
1590 LI=LI+10:RETURN
1600 LI=3100+(PN+A)*100:? #1;LI;"REM** MISSI
     LE ";PN;":RESOLUTION,MISSILE LENGTH,SIZ
     E. ";
1610 ? #1;"SUBSEQUENT LINE IS MISSILE DATA:G
     OS.LA"
1620 ? #1;LI;"DATA ";RE;",";VL;",";SI:GOSUB
     LA:GOTO 1560
9000 FOR I=1536 TO 1706:READ A:POKE I,A:NEXT
```

```
       I:RETURN
9010 DATA 162,3,189,244,6,240,89,56,221,240,
     6,240,83,141,254,6,106,141
9020 DATA 255,6,142,253,6,24,169,0,109,253,6
     ,24,109,252,6,133,204,133
9030 DATA 206,189,240,6,133,203,173,254,6,13
     3,205,189,248,6,170,232,46,255
9040 DATA 6,144,16,168,177,203,145,205,169,0
     ,145,203,136,202,208,244,76,87
9050 DATA 6,160,0,177,203,145,205,169,0,145,
     203,200,202,208,244,174,253,6
9060 DATA 173,254,6,157,240,6,189,236,6,240,
     48,133,203,24,138,141,253,6
9070 DATA 109,235,6,133,204,24,173,253,6,109
     ,252,6,133,206,189,240,6,133
9080 DATA 205,189,248,6,170,160,0,177,203,14
     5,205,200,202,208,248,174,253,6
9090 DATA 169,0,157,236,6,202,48,3,76,2,6,76
     ,98,228,0,0,104,169
9100 DATA 7,162,6,160,0,32,92,228,96
```

# Point Set Graphics

Douglas Winsand

*Explore an entire universe, move into it, starting from the simplicity of an innocuous mathematical expression. Best of all, you need not understand the underlying mathematics to voyage into these equations, but the explorations might well deepen your appreciation and knowledge of them.*

The computer can allow you to *see* mathematical events, to visualize the often delicate interactions of abstract, mathematical ideas. This program lets you generate, examine, and save pictures of recursive point sets.

These point sets create some very unusual computer graphics when plotted on your screen. You'll see misshapen, reversing spirals, abstract shapes, wisps of smoke, galaxies, and pointillist flowers – all composed of myriads of points. Most of these point structures are very hard to describe, some are quite beautiful. Many of them are also infinitely detailed. This program will turn your computer into a powerful microscope, allowing you to move closer into the detail of the inner structures in some of the point sets you'll generate.

What are recursive point sets? There are an infinite variety of ways to scatter points in a plane. At one extreme is the perfect straight order of points in a line. At the opposite extreme is a totally random scattering of points across the plane. Between these two extremes, there are an infinite number of collections or *sets* of points in the plane sets which are partially ordered and partially random. These are the point sets you'll be able to create.

The algorithm for generating recursive point sets is really quite simple. We begin with a seed point plotted in the x-y plane. We then plug the x and y coordinates of the seed point into a recurrence formula in order to generate the x and y coordinates of a new point. This new point is then used as the seed point in the recurrence formula to generate a third point and so on, *ad infinitum*.

A typical recurrence formula is:

$$X1 = Y - SIN(X)$$
$$Y1 = B^{*}X^{2} + X - 1$$

where

X1 is the x coordinate of the new point

Y1 is the y coordinate of the new point
X is the x coordinate of the seed point
Y is the y coordinate of the seed point

The recurrence formula is the heart of this process. By changing these equations, you can begin to create new recursive point sets.

## Running The Program

After you've entered the program (and made the appropriate changes if you use tape), RUN it. First you'll be asked whether you want to create a new point set, or view a point set which you have saved on tape or disk. If you enter a two, and you're using a disk drive, type in the name of the picture file you wish to see. If you are using tape, push PLAY when the computer beeps.

Since you haven't created any point sets yet, enter 1. The recurrence formula (lines 170-190) will be displayed. Type CONT and you'll see the points created by the recurrence formula begin to appear on the screen along with their x and y coordinates. The points will continue to be plotted until you take one of the following actions.

Either type C on the keyboard in order to change the recurrence formula, or type S to save a picture of a point structure which you've generated. When you type S, you'll be asked to supply an eight-character picture file name, if you're using a disk drive. (The computer will beep twice for tape. Push PLAY-RECORD.) The recurrence formula will be displayed in the text window so that you'll know how to create the same set later.

Or, typing M will stop the point set generator and display a flashing point which you can move around the screen with a joystick. Push the trigger on the joystick to freeze the flashing cursor point where it is. At whatever location you freeze the point, this will be the center of the screen when you again begin generating points. Using this feature, you can choose a point of interest and bring it to the center of the screen. While you are moving the cursor, you'll see two numbers displayed in the text window. These are the x and y coordinates of the cursor point. You can use these two numbers as references to specify areas within your point structures.

## Magnifying Beyond 100,000 X

Once you've frozen the cursor, you'll be asked to specify a *magnification factor*. If your point set is too large for the screen, enter a magnification factor between zero and one. The field of view will shrink by the specified amount. If you want to magnify the point set, enter a magnification factor greater than one. A rectangle will appear on the screen. The rectangle encloses an area which will occupy the entire screen when you begin generating points again. I suggest that you start with fairly low magnifications at first, and that you magnify in series of small steps rather than one large magnification. This is simply so that you won't lose sight of where you are in the point set. I've magnified point structures up to 100,000 times with no problem. Somewhere between a magnification of 100,000 and 1,000,000, however, the limited precision of your computer will become evident, and the computer will begin to randomize and destroy the order in your point structure.

After you've selected your magnification factor, you'll be given a choice of whether you wish to begin plotting points, starting with the last point you plotted before you called the magnification routine. I've included this last option to allow you to magnify point sets quickly. If you attempt a high magnification and begin plotting points from the original seed point, it may take a long time before the points begin to appear on the screen. There is also an S option at this point to allow you to save a point set with the magnification rectangle included. That's basically it.

If you let Program 1 run for about five minutes, you'll see a fluid, streamlined structure take shape. This point set is called the attractor of Henon. It turns out that each of the lines of points in the attractor of Henon is actually composed of several parallel lines of points, and each of those several lines of points is composed of several more parallel lines and so on, forever. For more information about attractors, let me refer you to Douglas Hofstadter's excellent article in the November 1981 issue of *Scientific American.*

To begin to see some of the detail in the attractor of Henon, type M and move the cursor point to (-.55,0), and then push the joystick trigger. Enter four for the magnification factor and then type B. Again, after about five minutes, you'll see an exploded view of the attractor. You'll also see the lines of points begin to resolve into separate parallel lines.

Generally, the higher the magnification, the longer it will take for a point set to develop a recognizable structure. I've frequently run a point set program for several hours (and sometimes overnight) in order to see more of the detail. Most of these point sets are composed of an infinite number of points, so you'll always see only partially completed sets. The more points in your set, the more it will look like the actual infinite set.

If you're using a cassette recorder, simply make the following changes:

1. eliminate lines 420 and 1160
2. change line 60 to: DIM B$(8):M=80:PRINT" ↰ ":PRINT:PRINT
3. change line 410 to: TRAP 430:LPRINT
4. change line 450 to: OPEN #2,8,0,"C:"
5. change line 1170 to: OPEN #2,4,0,"C:"
6. change line 1150 to: REM

## Explorer's Notes

Now that you know how to generate these point structures, you can begin to explore this strange territory.

As you try different recurrence formula and seed points, you'll find that, more often than not, the point structures your formula generates are divergent sets whose points quickly exceed the screen limits or the computer's numerical limits. If a point set diverges, vary the seed point, or change additive or multiplicative constants in the recurrence formula. Watch to see if one of these changes slows or quickens the divergence. Here is where a little perseverance will be rewarded.

Once you've found a convergent point set, you can usually create slightly different looking sets by making small variations in the same parameters mentioned above. Some of the point structures I've found will metamorphose into a totally different looking pattern when one of the parameters in the recurrence formula is changed. Try every kind of function that Atari BASIC offers. Using subroutines, I have tried Bessel and Legendre functions in some of my recurrence formulae.

The class of point sets which I've discovered most often are the spirals. Most of them slowly converge toward a center point, but I have found several spirals which slowy grow out from a center toward a fixed boundary. I have also run across several spiral structures which look like spiral galaxies.

Other types of point sets I've found are much harder to describe in a few sentences. I've included a few of these in the recurrence formulae in Figure 1. These will help to get you started, and give you an idea of the variety of point sets I've found. I'm sure there are many, many others.

For those of you whose curiosity is piqued, there are many new paths to explore. For example, with a few changes, each new point can be calculated from several preceding points rather than just one, immediately preceding, point. I've done a little experimenting using two preceding points to calculate a third, and I have found some convergent point structures.

For those of you with a printer or a movie camera, a striking animation sequence can be created by changing one of the parameters in the recurrence formula, then recording a picture, changing that same parameter by a small amount, recording a picture, and so on until you've built up about 20 or 30 of these pictures. Place these pictures in order and flip through them rapidly to create an animation effect. I've only had time to create one such sequence, but the effect is worth it. Lines of points appear to twist and writhe, evaporate back to points and later condense into lines again.

The concept underlying these point sets can be extended to three dimensions and, with a short algorithm, you can display these sets on your screen. You can even generate stereoscopic views of three-dimensional point structures.

**Figure 1**

```
1. X=0:Y=1
   X1=Y+1.4*X^2-1
   Y1=X^2-.3*X
2. X=0:Y=.58
   X1=Y+1.4*X^2-1
   Y1=X^2-Y^2+1
3. X=1:Y=.5
   X1=Y-(COS(X)^2)^1.25
   Y1=.01*X^2-X+1.125
4. X=1:Y=.5
   X1=Y-(COS(X)^2)^1.25
   Y1=.01*X^2-X+1.27
5. X=.01:Y=.001
   X1=2*Y*SIN(X)-1
   Y1=3*(COS(Y)-SIN(X))
6. X=.01:Y=.001
   X1=1.4*Y*SIN(Y)-X
   Y1=1.4*COS(Y)-.3*SIN(X)
7. X=.75:Y=.5
   X1=Y-(ABS(COS(X)-SIN(X)))^1.25
   Y1=.01*X^2-X+1.5
```

## PROGRAM. Point Set Graphics.

```
10 REM * RECURSIVE POINT SET UTILITY *
20 REM * DOUGLAS WINSAND - 1982 *
30 REM
40 REM * INITIALIZATION & OPTIONS *
50 REM
60 DIM A$(15),B$(8):A$="D1:{8 SPACES}.DAT":M
   =80:PRINT "{CLEAR}":PRINT :PRINT
70 PRINT "DO YOU WISH TO:"
80 PRINT "{7 SPACES}1) CREATE A POINT SET"
90 PRINT "{7 SPACES}2) VIEW A PREVIOUSLY STO
   RED SET"
100 INPUT B
110 IF B=1 THEN 320
120 IF B=2 THEN 1150
130 F=0:G=0:FLAG=0
140 B$="{8 SPACES}"
150 GRAPHICS 8:SETCOLOR 2,0,0:COLOR 1:OPEN #
    1,4,0,"K:"
160 REM * RECURRENCE FORMULAE *
170 X=0:Y=0:PLOT X,Y
180 X1=Y+1.4*X^2-1
190 Y1=0.3*X
200 REM * PLOT CALCULATED POINTS *
210 C=159+(X1-F)*M:D=79+(Y1-G)*M
220 IF C>0 AND D>0 AND C<319 AND D<159 THEN
    PLOT C,D
230 PRINT X1,Y1
240 REM * READ KEYBOARD & BRANCH *
250 IF PEEK(764)<>255 THEN GET #1,J:B$(1,1)=
    CHR$(J)
260 IF B$(1,1)="S" THEN GOSUB 410
270 IF B$(1,1)="M" THEN GOSUB 570
280 IF B$(1,1)="C" THEN GOTO 320
290 X=X1:Y=Y1:GOTO 180
300 REM
310 REM * CHANGE RECURRENCE RELATION *
320 GRAPHICS 0
330 PRINT :PRINT :PRINT
340 PRINT "CHANGE THE RECURSION FORMULA, THE
    N TYPE 'CONT' TO PLOT YOUR NEW POINT SET"
350 LIST 170,190
360 STOP
370 CLOSE #1:GOTO 140
380 REM
390 REM * STORE SCREEN ROUTINE *
400 REM
410 PRINT "ENTER 8 CHARACTER NAME OF NEW PIC
    TURE FILE.":INPUT B$:TRAP 410
```

```
420 FOR X=4 TO 11:A$(X,X)=B$(X-3,X-3):NEXT X
430 LIST 170,190
440 PICMEM=PEEK(88)+256*PEEK(89)
450 OPEN #2,8,0,A$
460 FOR J=PICMEM TO PICMEM+6400
470 IF PEEK(J)=0 THEN GOTO 490
480 P=(J-PICMEM)/256:PUT #2,INT(P):PUT #2,(P
    -INT(P))*256:PUT #2,PEEK(J)
490 NEXT J
500 PUT #2,0:PUT #2,0:PUT #2,0
510 TXTMEM=PEEK(660)+256*PEEK(661)
520 FOR J=TXTMEM TO TXTMEM+159:PUT #2,PEEK(J
    ):NEXT J
530 CLOSE #2:B$(1,1)=" ":RETURN
540 REM
550 REM * MAGNIFICATION SUBROUTINE *
560 REM
570 XO=159:YO=79
580 REM
590 REM * FLASHING CURSOR ROUTINE *
600 REM
610 POSITION XO,YO:PUT #6,1:H=STICK(0)
620 IF STRIG(0)=0 THEN GOTO 790
630 IF H<>15 THEN GOSUB 710
640 IF XO>319 THEN XO=XO-1
650 IF XO<0 THEN XO=XO+1
660 IF YO>159 THEN YO=YO-1
670 IF YO<0 THEN YO=YO+1
680 POSITION XO,YO:PUT #6,0
690 FOR L=0 TO 30:NEXT L
700 GOTO 610
710 POSITION XO,YO:PUT #6,0
720 IF H=6 OR H=10 OR H=14 THEN YO=YO-1
730 IF H=5 OR H=13 OR H=9 THEN YO=YO+1
740 IF H=6 OR H=7 OR H=5 THEN XO=XO+1
750 IF H=9 OR H=10 OR H=11 THEN XO=XO-1
760 PRINT "{CLEAR}"
770 PRINT INT((XO-159+M*F))/M,(-1)*INT((YO-7
    9+M*G))/M
780 RETURN
790 F=(XO-157+M*F)/M
800 G=(YO-79+M*G)/M
810 REM
820 REM * MAGNIFICATION ROUTINE *
830 REM
840 PRINT "ENTER MAGNIFICATION FACTOR":INPUT
     N
850 IF N<1 THEN GOTO 990
860 P1=XO-159/N:P2=XO+159/N:P3=YO-79/N:P4=YO
    +79/N
```

```
870 IF P2>319 AND P3<0 THEN PLOT P1,0:DRAWTO
     P1,P4:DRAWTO 319,P4:GOTO 990
880 IF P1<0 AND P3<0 THEN PLOT P2,0:DRAWTO P
    2,P4:DRAWTO 0,P4:GOTO 990
890 IF P1<0 AND P4>159 THEN PLOT 0,P3:DRAWTO
     P2,P3:DRAWTO P2,159:GOTO 990
900 IF P2>319 AND P4>159 THEN PLOT 319,P3:DR
    AWTO P1,P3:DRAWTO P1,159:GOTO 990
910 IF P1<0 THEN PLOT 0,P3:DRAWTO P2,P3:DRAW
    TO P2,P4:DRAWTO 0,P4:GOTO 990
920 IF P3<0 THEN PLOT P1,0:DRAWTO P1,P4:DRAW
    TO P2,P4:DRAWTO P2,0:GOTO 990
930 IF P4>159 THEN PLOT P1,159:DRAWTO P1,P3:
    DRAWTO P2,P3:DRAWTO P2,159:GOTO 990
940 IF P2>319 THEN PLOT 319,P3:DRAWTO P1,P3:
    DRAWTO P1,P4:DRAWTO 319,P4:GOTO 990
950 PLOT P1,P3:DRAWTO P2,P3:DRAWTO P2,P4:DRA
    WTO P1,P4:DRAWTO P1,P3
960 REM
970 REM * OPTIONS *
980 REM
990 B$(1,1)=" "
1000 PRINT "IF YOU WISH TO RESUME THE RECURS
     ION WHERE IT LEFT OFF PUSH 'R'."
1010 FOR L=0 TO 500:NEXT L
1020 PRINT "IF YOU WISH TO START THE RECURSI
     ON FROM THE BEGINNING,PUSH 'B'."
1030 FOR L=0 TO 500:NEXT L
1040 PRINT "IF YOU WISH TO SAVE THIS PICTURE
      PUSH 'S'."
1050 FOR L=0 TO 500:NEXT L
1060 IF PEEK(764)<>255 THEN GET #1,J:B$(1,1)
     =CHR$(J)
1070 IF B$(1,1)="R" THEN M=M*N:GRAPHICS 8:SE
     TCOLOR 2,0,0:RETURN
1080 IF B$(1,1)="B" THEN M=M*N:POP :GRAPHICS
      8:SETCOLOR 2,0,0:GOTO 170
1090 IF B$(1,1)="S" THEN FLAG=1:GOSUB 410
1100 IF FLAG=1 THEN FLAG=0:GOTO 1000
1110 GOTO 1060
1120 REM
1130 REM * RECALL STORED PICTURES *
1140 REM
1150 PRINT "ENTER 8 CHARACTER PICTURE FILE N
     AME":INPUT B$
1160 FOR X=4 TO 11:A$(X,X)=B$(X-3,X-3):NEXT
     X
1170 OPEN #2,4,0,A$
1180 GRAPHICS 8:COLOR 1:SETCOLOR 2,0,0
1190 PICMEM=PEEK(88)+256*PEEK(89)
```

```
1200 GET #2,J:GET #2,K:GET #2,L
1210 IF J=0 AND K=0 AND L=0 THEN 1230
1220 POKE J*256+K+PICMEM,L:GOTO 1200
1230 TXTMEM=PEEK(660)+256*PEEK(661)
1240 FOR X=TXTMEM TO TXTMEM+159:GET #2,J:POK
     E X,J:NEXT X
1250 CLOSE #2
1260 GOTO 1260
```

# Page Flipping

Rick Williams

*By changing only two bytes in the display list, you can cause the screen to display any portion of memory. This permits many interesting effects: coarse scrolling, instant screen fill, and page flipping, or "screen switching."*

Did you envy that article in **COMPUTE!**'s (November 1981, #18) Apple Gazette about page flipping? Well, relax. Atari can do it, too. Actually, there are two ways of doing it. The first way is a USR routine. I will get to the second way later.

I have written a machine language routine to transfer RAM to the screen. This routine (Program 1) will access BASIC through the USR function. To access this function, command "A=USR(1536,X)", 1536 being the start location of the machine program, and X being the location to start reading from RAM that will later be transferred to the screen. I have written the program only to work on graphics zero. Without this, strange effects will be seen on your TV.

Now on to the second way. This is the better way because it uses the horizontal scroll register. It scrolls into an entirely new frame. It works just as the machine routine. Write the high and low byte to the variables *L* and *H*. Here is the program:

## PROGRAM 1. Page Flipping.

```
1 GOTO 50
5 REM THIS REQUIRES NO GRAPHICS 0 COMMAND
10 DLIST=PEEK(560)+256*PEEK(561):POKE 82,0:R
   EM SET LEFT MARGIN
20 L=DLIST+4
25 H=DLIST+5
30 A=ADR(A$):B=INT(A/256):C=A-(256*B)
31 D=ADR(B$):E=INT(D/256):F=D-(256*E)
32 POKE L,C:POKE H,B:FOR I=1 TO 500:NEXT I
33 POKE L,F:POKE H,E:FOR I=1 TO 500:NEXT I:I
   F PEEK(764)=255 THEN 32
34 GRAPHICS 0:END
50 DIM A$(960),B$(960)
55 A$=CHR$(3):A$(960)=CHR$(3):A$(2)=A$:REM F
   ILL A$ WITH CHR$(3)
56 B$=CHR$(4):B$(960)=CHR$(4):B$(2)=B$:REM F
   ILL B$ WITH CHR$(4)
60 GOTO 10
```

## PROGRAM 2. Page Flipping.

```
10 REM *** PAGE FLIPPING BASIC ***
11 REM *** By Rick Williams
12 REM
15 DIM A$(50),B$(50)
20 FOR I=1536 TO 1536+42:READ A:POKE I,A:NEX
   T I:REM LOAD MACHINE CODE
21 GRAPHICS 0:FOR I=1 TO 50:A$(I,I)=CHR$(RND
   (1)*255):NEXT I:REM LOAD STRINGS
22 FOR I=1 TO 50:B$(I,I)=CHR$(RND(1)*255):NE
   XT I
23 GRAPHICS 0:A=USR(1536,ADR(A$)):FOR I=1 TO
    500:NEXT I
24 GRAPHICS 0:A=USR(1536,ADR(B$)):FOR I=1 TO
    500:NEXT I:GOTO 23
25 REM *** YOU MUST HAVE THE GRAPHICS 0 COMM
   AND TO RESET THE DISPLAY LIST
26 REM *** DATA FOR MACHINE CODE ***
1000 DATA 104,104,133,205,104,133,204,32,20,
     6,32,20,6,32,20,6,32,20,6,96,160,0,162,
     0,169,0,177,204,145,88,200
1010 DATA 152,240,3,76,26,6,230,89,230,205,9
     6,16
```

# An Introduction To Display List Interrupts

Alan Watson

*Many startling effects are possible with display list interrupts. This tutorial will get you started.*

Have you ever wondered how some commercial programs for your Atari display more than the four colors you can get from BASIC? It's done by using display list interrupts. In fact, it is possible to get all of Atari's 128 colors on the screen at the same time! While few programs ever call for all 128 colors, it is nice to see how it can be done.

Inside your Atari there is an integrated circuit which Atari calls ANTIC. This circuit takes care of the television screen display so the main processor can do other things in the program. ANTIC is a microprocessor and has its own program which it follows to display information on the screen. Its program is called the display list. The display list is different for each Graphics mode since each Graphics mode has different types and amounts of information which need to be displayed.

When a program encounters a Graphics command, the starting address for the display list is placed at decimal locations 560 (low byte of the address) and 561 (high byte). ANTIC looks at this address to find out what it needs to do. If no changes have been made in the display list, the first three instructions cause ANTIC to blank the first 24 lines. Since televisions overscan, this insures all our data will be in the visible area of the screen.

The next instruction (followed by an address) tells ANTIC where to find the display memory. Then comes instruction register (IR) mode bytes (see Table 1). The number of IR mode bytes depends on the Graphics mode that has been selected. Finally there is an instruction (followed by an address) to return to the start of the display list and start all over again.

**Table 1**

| BASIC GRAPHICS MODE NUMBER | IR MODE BYTE (DECIMAL) |
|---|---|
| 0 | 2 |
| 1 | 6 |
| 2 | 7 |
| 3 | 8 |
| 4 | 9 |
| 5 | 10 |
| 6 | 11 |
| 7 | 14 |
| 8 | 16 |

## The Three Necessary Steps

Creating a display list interrupt involves three steps. First, we must alter the IR mode byte for the line prior to the one we want to change. We do this by adding 128 (decimal) to it. Second, we must write a routine which tells the 6502 what we want to do during the interrupt. The third step is to allow the interrupt to happen by "enabling NMI" or POKEing decimal location 54286 with 192 (decimal).

When an interrupt occurs, the 6502 looks at decimal location 512 and 513 to find the address where interrupt instructions are located. The address is stored low byte then high byte. The examples for this article all use page six (starting at decimal location 1536) so each example POKEs 512 with zero and 513 with six.

Since we will interrupt the main processor to perform these instructions, we will have to save any registers we use and then restore them just before we return from the interrupt.

Here is how the interrupt display list program flows.

```
.
V
V
IR MODE BYTE
IR MODE BYTE + 128 > INTERRUPT VECTOR ADDRESS
                                 V
IR MODE BYTE        < INTERRUPT ROUTINE
IR MODE BYTE
      V
      V
```

Atari uses a number of registers (memory locations) to determine what colors and luminances should be used for background, plotted points, and characters. For each of these items there are a hardware register and a corresponding shadow register. Hardware registers are "write only" and cannot be read. They are updated from their respective shadow registers at the end of each frame (during the vertical blanking interval).

POKEing the shadow register is a quick alternative to the BASIC SETCOLOR command. You simply choose the color you want from the 16 colors listed on page 50 of *Atari 400/800 Basic Reference Manual,* multiply it by 16, and add the luminance value desired. Then POKE the result into the appropriate shadow register. (See Table 2.)

Table 2

| TO SET THE COLOR OF ... | POKE DESIRED VALUE INTO | CORRESPONDING HARDWARE REGISTER |
|---|---|---|
| Plotted Points Using COLOR 1 | 709 | 53271 |
| Plotted Points Using COLOR 2 | 710 | 53272 |
| Plotted Points Using COLOR 3 | 711 | 53723 |
| Background | 712 | 53274 |

Note: This table is for Graphics modes 1 through 7.

If we place the change resulting from our display list interrupt in the appropriate hardware register, the part of the screen below the interrupt will change. The top of the screen remains as it was because the hardware register is updated from its shadow register at the end of the frame.

Now that you have the idea, type in Program 1 and RUN it.

## Background Luminance, And More

All that changes is the background luminance. But you can do lots more with this example!

Let's examine the interrupt routine we used.

| Memory Location Used in Example | Decimal Value in Data Statement (Line 150) | Assembly Language Mnemonic | Comments |
|---|---|---|---|
| 1536 | 72 | PHA | Save accumulator |
| 1537 | 169 | LDA | Load accumulator |
| 1538 | 6 | #6 | with new color |
| 1539 | 141 | STA | Wait for horizontal |
| 1540 | 10 | $0A | so change doesn't |
| 1541 | 212 | $D4 | occur in mid line |
| 1542 | 141 | STA | Store new color |
| 1543 | 26 | $1A | in hardware |
| 1544 | 208 | $D0 | register |
| 1545 | 104 | PLA | Restore accumulator |
| 1546 | 64 | RTI | Return from interrupt |

You can choose any color you like for the bottom by determining its value (the same way as I mentioned above) and using it as the third number in the DATA statement (line 150). I've listed the memory locations above and left of a text window. I did this so you can type in POKE commands to make changes and watch to see what happens. To change the bottom color, POKE 1538 with the new color value you want.

You can choose any color you like for the top of the screen as well by POKEing the value into shadow register 712.

What's more, you can change the top and/or bottom color of any of the plotted rectangles. To change them in the direct mode (after the program has run and the "READY" prompt appears), POKE the appropriate shadow register with the desired color value for the top and POKE the corresponding hardware register with your color choice for the bottom.

See Table 2 to determine which registers to use for each of the rectangles. The left rectangle is plotted using COLOR 1, the center using COLOR 2, and the right using COLOR 3.

If you would rather make the changes in the program itself, change the eighth number in the DATA statement (line 150) to the low byte of the appropriate hardware register. For the rectangle using COLOR 1 use 22, for COLOR 2 use 23, and for COLOR 3 use 24.

To select a different vertical position for the color change to occur, add 128 to a different display list instruction. We've been using the instruction as START + 24 (line 240) which places the interrupt midway down the screen. Since there are 40 display blocks in BASIC Graphics mode 5, you can experiment anywhere from START + 6 to START + 44 without problems.

Once you understand how to create and use display list interrupts, your programming capabilities are expanded. You can use them together with player-missile graphics to change player color, width, and/or horizontal position as the player passes through the interrupt line. With a slightly more complicated interrupt routine, a player can be drawn with the same or a different shape several times at different vertical positions on the screen.

A good example of this can be found in John Palevich's article "Shoot" in the September, 1981, issue of **COMPUTE!** Magazine (page 86). When programming with text modes, you can change character sets in mid screen. Atari's "Space Invaders" uses many display list interrupts. Many things are possible, and you'll discover more as you experiment.

## All 128 Colors At Once

Atari is capable of showing all 128 colors at the same time. There are a number of ways this can be done. Program 2 shows one way. The two biggest changes compared to Program 1 are the interrupt routine and the custom display list.

The interrupt routine is written so that each time it's used, the color/luminance value is increased by two. By the end of the display, 128 colors appear.

A custom display list was created to get more than enough vertically displayed lines for the 128 colors and to be able to plot a design. BASIC Graphics 8 is essentially a one-color mode (one color/luminance and one luminance are available). Changes in the background show up in the plotted points as well. But, between BASIC Graphics 7 and Graphics 8, there is an ANTIC instruction register mode 14 (decimal) which has the same number of vertical positions as BASIC Graphics 8. It has only half as many horizontal positions. What you get instead is a four color mode. Program 2 uses only two of the four registers available!

In addition to the references cited at the end of this article, credit and thanks go to Judy Bogart at Atari, who explained how hardware registers are updated and helped with the interrupt routine in Program 1. It was my first attempt to use assembly language in a program.

## References:

*Atari 400/800 Basic Reference Manual.* Atari, Inc., copyright 1980.

*Atari Personal Computer System Hardware Manual.* Atari, Inc., copyright 1980.

**PROGRAM 1. An Introduction To Display List Interrupts.**

```
10 REM *** An Introduction To Display List I
   nterrupts
20 REM *** Listing #1
30 REM *** Alan Watson
40 REM *** Nov. 9, 1981
100 REM *** POKE CODE INTO PAGE 6
110 FOR I=0 TO 10
120 READ C
130 POKE 1536+I,C
140 NEXT I
150 DATA 72,169,6,141,10,212,141,26,208,104,
    64
160 REM *** POKE INTERRUPT VECTOR ADDRESS
170 POKE 512,0:POKE 513,6
200 REM *** GRAPHICS CALL AND FIND DISPLAY L
    IST
210 GRAPHICS 5
220 START=PEEK(560)+256*PEEK(561)
230 REM *** MODIFY DISPLAY LIST IR MODE BYTE
240 POKE START+24,10+128
300 REM *** PLOT SOMETHING ON THE SCREEN
310 FOR X=1 TO 3
320 COLOR X
330 PLOT 20*X+5,30:DRAWTO 20*X+5,10
340 DRAWTO 20*X-5,10:POSITION 20*X-5,30
350 POKE 765,X
360 XIO 18,#6,0,0,"S:"
370 NEXT X
380 REM *** SHORT DELAY BEFORE COLOR CHANGE
390 FOR D=1 TO 300:NEXT D
400 REM *** ENABLE NMI
410 POKE 54286,192
```

**PROGRAM 2. An Introduction To Display List Interrupts.**

```
10 REM *** An Introduction To Display List I
   nterrupts
20 REM *** Listing #2
30 REM *** Alan Watson
40 REM *** Nov. 9, 1981
100 REM *** POKE CODE INTO PAGE 6
110 FOR I=0 TO 17
120 READ B
130 POKE 1536+I,B
140 NEXT I
150 REM *** POKE INTERRUPT VECTOR ADDRESS
160 POKE 512,0:POKE 513,6
200 REM *** GRAPHICS CALL AND FIND DISPLAY L
    IST
210 GRAPHICS 8
220 START=PEEK(560)+256*PEEK(561)
230 COLOR 1
240 FOR I=1 TO 5
250 READ X1,Y1,X2,Y2,X3,Y3,X4,Y4
260 PLOT X1,Y1:DRAWTO X2,Y2:DRAWTO X3,Y3:POS
    ITION X4,Y4
270 POKE 765,1
280 XIO 18,#6,0,0,"S:"
290 NEXT I
300 REM *** CREATE CUSTOM DISPLAY LIST (IR M
    ODE 14)
310 POKE START+3,78
320 FOR I=6 TO 33:POKE START+I,14:NEXT I
330 FOR I=34 TO 98:POKE START+I,14+128:NEXT I
340 POKE START+99,78+128
350 FOR I=102 TO 164:POKE START+I,14+128:NEXT I
360 FOR I=165 TO 198:POKE START+I,14:NEXT I
370 POKE START+199,65
380 POKE START+200,80
390 POKE START+201,128
400 REM *** ENABLE NMI
410 POKE 54286,192
800 REM *** CODE FOR INTERRUPT ROUTINE
810 DATA 72,173,198,2,24,105,2
820 DATA 141,10,212,141,198,2,141,24,208,104
    ,64
830 REM *** PLOT POINTS
840 DATA 139,155,116,27,77,27,100,155
850 DATA 219,155,242,27,203,27,179,155
860 DATA 189,99,160,35,158,35,129,99
870 DATA 140,155,159,99,130,99,139,155
880 DATA 179,155,189,99,159,99,178,155
```

# Extending Atari High Resolution Graphics

## Part 1: The Polygon Fill Subroutine

Phil Dunn

*In this three-part series, Mr. Dunn introduces the reader to advanced graphics techniques, from a flexible fill routine to realistic "textured" graphics, and concludes with a technique that allows more than 66 pseudo-colors in Graphics Mode 8.*

*Polyfil is a versatile subroutine that permits filling in any shape, not just boxes and trapezoids, as with the XIO Fill command.*

This is the first of three essays which will develop some methods to aid in creating more dramatic displays in the Atari high resolution graphics modes. The intent is to show how we can use the hi-res modes more effectively, and to set up techniques that enable us to do these things in the easiest possible way.

One very useful function that is not in the Atari BASIC repertoire is the polygon fill subroutine. The ease with which this function enables us to create pictures with large complex shapes is so significant that any graphics system lacking it should probably be considered incomplete.

A type of color fill is available within the Atari BASIC function options. However, its limitations are such that it provides very little convenience for developing pictures with shapes and areas that have many angles and irregular boundaries. Naturally, the most interesting pictures usually have just this characteristic!

The example program shows the power of the polygon colorfill subroutine, Polyfil. Let's scan the program to get an overview of how it works. Lines 200 to 240 establish Graphics mode 7 without the text window and set the four color registers (that we are allowed) to the colors that we choose. Lines 250 to 400 set up the numeric values and transfer them to Polyfil, which starts at line 17000. The actual data used by the Polyfil subroutine to create this particular picture are on lines 420 to 730. You can use this program to create your own scene by changing these lines.

Lines 800 to 959 have nothing to do with the polygon fill. They set the images of two figures in the usual PLOT-DRAWTO way. You may note that there is almost as much coding in lines 800 to 955 for these two little figures as there is in lines 430 to 730 for the rest of the entire picture done with Polyfil.

## A Programming Trick

Incidentally, there is a little programming trick used here that you might want to note. I was not sure of the exact screen locations that would most effectively, aesthetically locate the figures. So I calculated the X, Y pixel locations relative to a reference pixel at the head of each figure. Then that number is added to the variable XX or YY for the PLOT-DRAWTO commands. I just changed the XX and YY values until the image looked appropriate. The same thing could have been done in lines 350 and 360, to add bias numbers to the X, Y values for the area vertex points if I weren't sure of the placement of these large areas.

Lines 16000 to 16130 just describe the input and variable requirements for the Polyfil subroutine. The actual subroutine itself only consists of lines 17000 to 17190. This subroutine was "mashed" for minimum memory usage, and then carefully "unmashed" for ease in transcribing. If you want to type it in as a mashed version, all you have to do is to chain all the lines that are not even multiples of ten into the previous line that is. Thus, lines 17001, 17002, 17003, 17004, and 17005 can all be chained into line 17000. Similarly, lines 17020 and 17021 can be combined, and lines 17030, 17031, 17032, 17033, and 17034 can all be placed on the same line.

## How To Set Up POLYFIL

Now let's get into the input requirements for the Polyfil subroutine. The first value it must have is the number of vertex points around the perimeter of the polygon, entered in the variable NP. For a triangular area, this number would be three, for a rectangle it would be four, etc. Then it needs the X, Y values for each perimeter vertex point.

The user must have previously DIMensioned the array variables X(), Y(), R(), and S(). The DIMension size must be equal to or greater than the number of vertex points. The vertex point values must be loaded into the array elements, starting with X(1), Y(1). Polyfil then proceeds to fill the area with the usual PLOT-DRAWTO commands. It should be noted that Polyfil uses the variable names Q1, Q2, I, J, YMAX, YMIN, YNOW, IM, IP, XA, and XB.

For those who want to understand how Polyfil works, an explanation follows. For each line segment around the perimeter, the slope and Y axis intercept are calculated in lines 17020 and 17021, respectively. Then the maximum and minimum Y axis values of the perimeter vertex points are determined in lines 17034 and 17040. The initial starting point is established as the smallest Y axis value on line 17051.

The Y value is then increased by one on line 17130 and the intercept points on the horizontally opposite sides are calculated as XA and XB on lines 17090 and 17110. A horizontal line is then drawn connecting the two points on line 17120. Y is then increased again and this is repeated until Y is the maximum polygon value. There is additional logic for vertical lines where the slope is not calculable, and to switch intercept calculations from one line segment to another as vertex points are passed over.

Now let's go over the program in some detail so we can see exactly what is done to use the Polyfil routine. First, an explanation of those code numbers in the data. The first number in each DATA statement assigns a color register to that shape. So, the mountain data on line 430 is assigned to color register 2. The second number specifies the number of vertex points in the shape. The mountain has six vertex points. The remainder of the data consists of the X, Y values for each vertex point. So, the first specified vertex point for the mountain is the X, Y values of 54,42, the second vertex point is 68,30, etc.

It should be noted that, when establishing these areas for Polyfil, the farthest background areas should be defined first and the nearest foreground areas should be defined last. That's how you get the visual effect of distance, where the "near" object is drawn over the "far" object.

There are only two limitations on how the vertex point data must be specified for Polyfil. The first restriction is that the points must be specified in a sequential order around the polygon perimeter. The second restriction is that the polygon shape must not have any indentations that require the fill technique to skip over empty spaces. Since the fill method used here consists of a series of horizontal lines, this means that "E" and "L" type shapes are ok, but "U" and "H" type shapes will not be filled properly. This is no major limitation, since any unacceptable shape can always be split up into two or more acceptable shapes, each of which can be filled separately.

Now, back to the essential program factors for using the Polyfil subroutine. Line 250 DIMensions the variables X(), Y(), S(), T().

Line 320 READs the first two bytes of DATA and assigns them to the variables COLR and NP, respectively. However, if it READs a -1 value for COLR, that signifies the end of DATA, and the program then jumps to line 800. Lines 330 to 370 then use that NP value to READ the next NP amount of number-pairs and store them into the X(), Y() arrays. Line 380 then applies the value in the COLR variable to set the COLOR command. Finally, line 390 GOSUBs to the Polyfil subroutine which does the job.

By now you may have loaded the example program and seen the picture. Within its limitations, it "works" the way I intended it to. Not only is it a good example for using the Polyfil subroutine, but it has aesthetic quality. Within the limitations of the graphics techniques we are using here, the picture is ok.

However, because of these limitations of Graphics Mode 7, only four colors, and slightly rough resolution relative to Mode 8, the picture seems somewhat flat and blocky.

Of course, we could spruce it up a little. We could get more colors by replacing the figures and the tree trunks with player-missile images. We could use David Small's Display List Interrupt driver routine to assign different colors to our registers at different vertical heights down the screen.

All these things could be used to improve the picture to a certain extent. However, the picture would still have a somewhat blocky and flat appearance.

The next essay on "Extending High Resolution Graphics" will address some aspects of this problem.

## PROGRAM. The Polygon Fill Subroutine.

```
100 REM ==============================
105 REM ={4 SPACES}POLYFIL Subroutine
    {5 SPACES}=
110 REM =  Demonstration  Program{3 SPACES}=
120 REM ==============================
125 REM ={11 SPACES}by:{13 SPACES}=
130 REM ={8 SPACES}Phil Dunn{10 SPACES}=
135 REM ={6 SPACES}12 Monroe Ave.{7 SPACES}=
140 REM ={3 SPACES}Hicksville, NY 11801
    {4 SPACES}=
150 REM ==============================
170 REM
200 GRAPHICS 7+16
210 SETCOLOR 0,5,6:REM C.1 = MAROON
220 SETCOLOR 1,10,8:REM C.2 = GREEN
230 SETCOLOR 2,9,10:REM C.3 = WHITE, WINDO
240 SETCOLOR 4,8,8:REM C.0 = BLUE, BACKG.
250 DIM X(12),Y(12),S(12),T(12)
280 POLYFIL=17000
305 REM Read areas and fill them in
315 REM First read the color & no. of points.
320 READ COLR,NP
322 IF COLR=-1 THEN 800
325 REM Now read in all the vertex point coo
    rdinates
330 FOR N=1 TO NP
340 READ XX,YY
350 X(N)=XX
360 Y(N)=YY
370 NEXT N
375 REM Now set the color
380 COLOR COLR
385 REM Now let the subroutine fill it in
390 GOSUB POLYFIL
400 GOTO 320
402 REM ==============================
403 REM ={4 SPACES}Scene from the book
    {4 SPACES}=
404 REM =  'Stranger By The River'  =
405 REM ={4 SPACES}by  Paul Twitchell
    {5 SPACES}=
406 REM ==============================
420 REM MOUNTAIN
430 DATA 2,6,54,42,68,30,85,22,110,35,118,45
    ,54,45
440 REM MOUNTAIN TOP
450 DATA 3,6,85,22,98,29,95,30,88,32,78,29,7
    4,27
```

```
470 REM LOWLANDS
480 DATA 2,5,0,44,80,40,159,44,159,45,0,45
520 REM FOREGROUND
530 DATA 2,4,0,65,159,65,159,96,0,96
540 REM RIVER
550 DATA 3,12,0,44,159,44,159,66,140,68,125,
    70,105,74,80,77,60,78,47,78,40,77,25,75,
    0,69
630 REM LEFT TREE TOP
650 DATA 2,12,25,25,37,34,32,35,36,40,30,43,
    33,55,18,59,5,50,12,41,8,37,15,32,12,30
660 REM RIGHT TREE TOP
670 DATA 2,8,130,32,149,39,145,48,153,56,135
    ,68,116,61,120,53,115,47
690 REM LEFT TREE TRUNK
700 DATA 1,6,11,80,16,56,19,52,21,57,17,81,1
    4,83
720 REM RIGHT TREE TRUNK
730 DATA 1,6,131,83,130,65,133,59,136,65,141
    ,82,136,85
750 DATA -1,-1
800 REM Paul
805 COLOR 0
810 XX=55
820 YY=80
825 PLOT XX-1,YY+3:DRAWTO XX-1,YY+8
830 PLOT XX,YY:DRAWTO XX,YY+8
835 PLOT XX+1,YY+3:DRAWTO XX+1,YY+8
840 PLOT XX+4,YY+4:DRAWTO XX+4,YY+8
845 PLOT XX+1,YY:PLOT XX+1,YY+1
850 PLOT XX+2,YY+4:PLOT XX+3,YY+4
855 PLOT XX+3,YY+5:PLOT XX+3,YY+6
860 PLOT XX+2,YY+6:PLOT XX+2,YY+7
865 PLOT XX+5,YY+8
870 PLOT XX+1,YY:PLOT XX+1,YY+1
880 PLOT XX+2,YY+4:PLOT XX+3,YY+4
900 REM Rebezar
905 COLOR 1
910 XX=65
915 YY=70
920 PLOT XX-1,YY+2:DRAWTO XX-1,YY+13
925 PLOT XX,YY:DRAWTO XX,YY+10
930 PLOT XX+1,YY:DRAWTO XX+1,YY+10
935 PLOT XX+2,YY+2:DRAWTO XX+2,YY+13
940 PLOT XX+5,YY+1:DRAWTO XX+5,YY+13
945 PLOT XX-4,YY+1:PLOT XX-3,YY+2
950 PLOT XX-2,YY+2:PLOT XX+3,YY+3
955 PLOT XX+4,YY+4
999 GOTO 999
16000 REM ==========================
16005 REM ={8 SPACES}POLYFIL{10 SPACES}=
16010 REM =A Polygon Fill Subroutine=
16020 REM ={5 SPACES}by Phil Dunn{8 SPACES}=
16025 REM ============================
16030 REM Enter with the values...
16040 REM NP = No. of vertex points.
16050 REM X = DIM array of X values
16060 REM Y = DIM array of y values
16070 REM S = DIM array used here
16080 REM T = DIM array used here
16090 REM Uses variables...
16100 REM Q1,Q2,I,J,YMAX,YMIN,YNOW
16110 REM IM,IP,XA,XB
17000 Q1=1
17001 Q2=1000
17003 FOR I=Q1 TO NP
17004 J=I+Q1
17005 IF J>NP THEN J=Q1
17010 IF X(I)=X(J) THEN S(I)=Q2:GOTO 17030
17020 S(I)=(Y(J)-Y(I))/(X(J)-X(I))
17021 T(I)=Y(I)-S(I)*X(I)
17030 NEXT I
17031 YMAX=-Q2
17032 YMIN=Q2
17033 FOR I=Q1 TO NP
17034 IF YMAX<Y(I) THEN YMAX=Y(I)
17040 IF YMIN>Y(I) THEN YMIN=Y(I):J=I
17050 NEXT I
17051 YNOW=YMIN
17052 IM=J-Q1
17053 IF IM<Q1 THEN IM=NP
17060 IP=J+Q1
17061 IF IP>NP THEN IP=Q1
17070 GOTO 17130
17080 IF S(J)=Q2 THEN XA=X(J):GOTO 17100
17090 XA=(YNOW-T(J))/S(J)
17100 IF S(IM)=Q2 THEN XB=X(IM):GOTO 17120
17110 XB=(YNOW-T(IM))/S(IM)
17120 PLOT XA,YNOW:DRAWTO XB,YNOW
17130 YNOW=YNOW+Q1
17131 IF YNOW<Y(IP) THEN 17160
17140 IF Y(IP)=YMAX THEN RETURN
17150 J=IP
17151 IP=IP+Q1
17152 IF IP>NP THEN IP=Q1
17160 IF YNOW<Y(IM) THEN 17080
17170 IF Y(IM)=YMAX THEN RETURN
17180 IM=IM-Q1
17181 IF IM<Q1 THEN IM=NP
17190 GOTO 17080
```

# Part 2: Textured Graphics

Phil Dunn

*This article extends the techniques covered in Part 1, the Polyfil subroutine. You'll be surprised at what "textured" graphics can do.*

With the Polygon Fill subroutine, we showed how easy it was to create a picture. However, the picture quality suffered because of the limitations of the Atari Graphics Mode 7 in terms of variety of color. Being limited to only four colors puts quite a constraint on our ability to develop an interesting picture. Here we are going to use some simple texture-creating effects.

Although we are restricting ourselves to the development of static pictures, this is just the necessary initial work that we can later build upon as we move into the creation of dynamically changing scenes with action, movement, and sound.

## Polygon Fill Becomes PolyPaint

The key tool that we had developed in Polygon Fill we will now expand upon to do a much greater task. I call this new subroutine PolyPaint because it provides so much variety of possible textures and colors within any defined polygon shape.

First let's get an overview of what is being presented here. Program 1 shows the PolyPaint subroutine which is going to do all this picture-painting for us. It starts off with a long list of REM statements which summarize its requirements and abilities. We will go over these in detail in the following paragraphs.

With all REM statements deleted, these programs will run with only a 16K memory system up to Graphics Mode 7.

Program 2 is the Picture program. Although the listing does not show it, it must be understood that the PolyPaint subroutine is to be attached to this program before it can be RUN. All that is needed in order to use this program will be explained in the remainder of this article.

Program 3, Palette, enables us to study the effects of various command options, pattern characteristics, hue-luminance assignments, and resolutions in the various graphics modes. Naturally, this too requires the PolyPaint subroutine to be appended before it will RUN.

Now let's take a close look at Program 1, the PolyPaint subroutine. Starting at the top, there are the REM statements which summarize the requirements and capabilities of PolyPaint. The first item mentioned is the variable NP, which must have the value of the number of vertex points around the perimeter of the polygon. The actual values for the X,Y coordinates of these vertex points are stored in the array variables X() and Y(). These variables are to be DIMensioned by the program that GOSUBs to PolyPaint, and the vertex points are to be stored in them with the first point being in X(1), Y(1), the second point in X(2), Y(2), etc.

## Vertex Data Point Restrictions

There are some restrictions upon how the vertex data point values may be specified. First, they must be defined in a sequential order around the polygon perimeter. Second, there can be no indentations that require the fill technique to skip over empty spaces. The previous Polygon Fill subroutine used a series of horizontally drawn lines to fill in the polygon. That method prevents trouble filling in "L," "E" or "F" shapes, for example, but it cannot fill in a "U" shape properly.

The same restriction exists for this PolyPaint routine when the option selected consists of either horizontal bars, or horizontal pixel sweep. On the other hand, if PolyPaint is used with the vertical bars or vertical pixel sweep options, it would have no trouble filling in the "L" or "U" shapes, but it would not fill the "E" or "F" shapes properly. Naturally, any indented shape can be divided into two or more non-indented shapes, which can then be filled properly by any option.

We have jumped ahead of ourselves by referring to the options before this point, but now we will cover the various options that are available by assigning the appropriate value to the variable called TYPE. When TYPE=1, the PolyPaint routine will allow us to "paint" our polygon with bars, or lines. I prefer to use the word *bar* here, to distinguish this option from the option available when TYPE=4.

When TYPE=1 the polygon is painted a whole bar at a time with the PLOT-DRAWTO commands. When TYPE=2 the polygon is painted a single pixel at a time using only the PLOT command. When TYPE=3 the polygon is painted a single pixel at a time also, but only with a specific checkerboard pattern.

The TYPE=4 option is a special feature that has nothing to do with polygon filling at all, but is very useful for drawing pictures in general. The TYPE = 4 option allows us to draw a line a pixel at a time, where the pixel can alternate between two colors.

If we set TYPE = 1 for bar painting, then we must assign values to four additional variables to specify how these bars will be used. The first variable, DIR, specifies the direction of the bar orientation. A value of DIR = 0 provides horizontal bars, and a value of DIR = 1 provides vertical bars. The variable SPA sets spaces between the bars. When SPA = 0 there are no spaces, when SPA = 1 there is one space between each bar, when SPA = 2 there are two spaces between each bar, etc. "FAC" is a variable space factor. When FAC = 0, the spaces between the bars (if any) are constant across the polygon. When FAC = 1 the spaces between the bars increase as they are placed within the polygon. When FAC = 1 the spaces between the bars decrease as they are drawn.

The bars are always drawn from the low value of the screen variable to the high value. Thus, the first horizontal bar will be drawn at the lowest value of Y, the top of the polygon, and the last horizontal bar will be drawn at the largest value of Y, at the bottom of the polygon. Similarly, vertical bars will be drawn from left (the smallest X) to right (to the largest X).

The last variable that must be assigned a value for bar painting is CLO. This value is used to set the color register for the bars. All the bars are drawn with the color of the CLO register.

When we set TYPE = 2 for pixel painting, this too requires four additional variables to be set. The DIR variable is used to determine the pixel "sweep" direction. The pixels are drawn in a sweep of one line at a time. When DIR = 0 the sweep lines are horizontal, and when DIR = 1, the sweep lines are vertical, much like the bar painting convention.

The FAC variable is used to specify the color – blending characteristic for the two color registers that are specified in the variables CLO and CHI. There are two different blending techniques that may be used; the FAC variable is used to specify which of the two techniques will be used and how the technique will be applied.

When FAC>0.0 and FAC<1.0, the polygon will be filled evenly with a proportional blend between the two colors of CLO and CHI, where the color of each pixel is selected at random with a probability determined by the value of FAC. The smaller the value of FAC the more the CLO color will predominate, and the higher the value of FAC the more the CHI color will be represented. FAC is the percentage of the CHI color. When FAC = .2, there will be 80% of the CLO color, and 20% of the CHI color. When FAC = .5, there will be a 50-

50 mix between the two colors. When FAC=.9, there will be 10% of CLO and 90% of CHI.

## Creating Shading Effects And Multiple Overlays

When FAC is equal to or greater than 1.0, a completely different color blending technique is used. With this technique, the CLO color always predominates at the end of the polygon with the low value for the screen variable, and the CHI color always predominates at the high value end of the polygon. For horizontal sweep (DIR=0), the CLO color will predominate at the top of the polygon and the CHI color will predominate at the bottom of the polygon. For vertical sweep (DIR=1), the CLO color will predominate at the left side of the polygon and the CHI color will predominate at the right side. (This is useful for shading effects.)

The rate at which the color ratio changes from CLO to CHI as the polygon is filled is determined by the value of FAC. When FAC is small (equal to or close to one), then the CLO color will dominate the polygon until the very end, when some amount of CHI color will appear. When FAC is large (ten or more), some amount of CLO color will appear in the beginning, but the CHI color will rapidly take over and will dominate the finished polygon. A value of FAC=5 will provide a fairly even balance between CLO and CHI, with the center of the polygon being about a 50-50 mixture.

This pixel painting technique allows the possibility of multiple color overlays. If the color register number assigned to CLO or CHI is -1, that color becomes the "transparent" color. In other words, that pixel is passed over and its color remains unchanged. Thus, the same polygon can be painted several different times; each time the transparent color will allow the previously drawn pattern to show through.

Polygon painting TYPE=3 provides a checkerboard pattern that alternates between CLO and CHI. The checkerboard may be painted by sweeping horizontally (DIR=0) or vertically (DIR=1). In the high resolution mode of Graphics 7, the checkerboard pattern is sufficiently small so that it can be used as a whole new color. Thus, by blending the allowable four colors of Graphics 7 two at a time in the checkerboard pattern, we can obtain six more colors. This gives us ten colors in Graphics Mode 7! This color blending technique works best when the two colors used have about the same luminance level. When the luminance level is vastly different between the two colors, the checkerboard characteristic becomes more visually obvious.

The final option found in this routine, TYPE=4, does not fill a polygon at all, but simply draws a line. The starting point is specified by variables X(0), Y(0) and the ending point by X(1), Y(1). However, this option allows the pixel line elements to alternate between the color of the CLO register and the color of the CHI register. Naturally, the same register number can be used for both CLO and CHI to plot a line of that one color.

If you plan to use this routine in a program of your own, you should make note of the variable names that are used internally here. These names are given in the REM statements in lines 17295 to 17305.

The structure of the PolyPaint routine is a direct expansion from the structure of the Polygon Fill subroutine presented previously. Many of the lines are identical. The routine uses the polygon perimeter vertex points to calculate the slope and Y-axis intercepts of the polygon perimeter line segments, and stores this information in the array variables S() and T() in line 17335. The maximum and minimum polygon values are calculated in lines 17345 to 17360. The intercept points for horizontal bars or pixel-sweep are calculated in lines 17395 to 17410, and for vertical bars or pixel-sweep, in lines 17420 to 17435.

Bar painting is done in lines 17455 to 17470, checkerboard painting is done in lines 17495 to 17515, and pixel painting is done in lines 17520 to 17595. The sweep line, or bar, is incremented in line 17600. At lines 17615 to 17650 the incremented line is checked to find if it passed over an adjacent vertex point. If it did, that point is checked against the maximum point on the polygon. If it is not beyond the maximum, the index pointer to the perimeter slope-intercept array elements is incremented.

So much for the PolyPaint subroutine itself. Now let's take a look at how it may be used. The example program (Program 2) is called the Picture program. This program has been written in a very general and useful form. To use it to create your own pictures, just redefine the Graphics mode and SETCOLOR assignments in lines 240 to 280, and the DATA and PLOT-DRAWTO commands after line 1020. The rest of the program is designed to read the DATA statements that are set up and let the PolyPaint subroutine do its job. Error detecting logic has been coded into this program so that if a mistake is made in setting up the DATA statements, the chances are that the program will catch it and tell us what it is and where it is located.

The PICTURE program sets up the necessary array variables in line 300. No polygon used here has more than 15 vertex points. Line

310 establishes the first statement which can be executed in the PolyPaint subroutine as being at line 17315. The polygon or area counter variable, A, is set to zero at line 370 and then the first data value is read in.

The DATA in lines 1080 to 1370 is arranged in a very specific way to minimize the difficulty of reading and debugging it. The first number in every DATA statement corresponds to the TYPE variable for the PolyPaint subroutine. The number should only be from one to four, depending upon the TYPE of painting that is desired. The only exception to this is the very last DATA statement where the TYPE value of 999 tells the PICTURE program that there is no more data. The value for TYPE determines the interpretation of the remaining numeric values on the remainder of each DATA statement.

When TYPE = 1, the next four DATA numbers represent the values for the PolyPaint subroutine variables DIR, SPA, FAC, and CLO, respectively. When TYPE = 2, the next four numbers represent the variables DIR, FAC, CLO and CHI. When TYPE = 3, the next three values represent the variables DIR, CLO and CHI.

The following DATA value for TYPE = 1, 2, or 3 is for the variable NP, and the remaining "NP" pair of values are for the vertex point array variables X(1), Y(1), X(2), Y(2), etc.

When TYPE = 4, the remaining six DATA values correspond to the variables CLO, CHI, X(0), Y(0), X(1), and Y(1), respectively.

## How The Picture Is Painted

Now let's take a look at how this particular picture is being painted. As you read this, you might want to RUN this program so you can see each area being painted as we mention it.

The first item painted is the mountain, in line 1080. It is first established as a TYPE = 3 checkerboard blend, using a DIR = 1 vertical sweep, and blending the two colors CLO = 3 (white) and CHI = 2 (green). This mountain is a polygon with NP = 6 vertex points, the first one being 54,42, the second 68,30, etc. Then, in line 1090, the left half of the mountain is repainted with TYPE = 2 pixel painting, DIR = 1 vertical sweep, FAC = 9 for mostly CHI color, and the color values CLO = 2 (green) and CHI = -1 (transparent). This gives a green shading effect at the left edge of the mountain. Line 1100 continues repainting the right side of the mountain with TYPE = 2 pixel painting, DIR = 1 vertical sweep, FAC = 2 for mostly CLO color, and CLO = -1 (transparent) and CHI = 2 (green). This gives a green shading effect to the right edge of the mountain.

On line 1120, the mountaintop is defined by a TYPE = 2 pixel painting form, with DIR = 0 horizontal sweep, FAC = 1 for mostly CLO color, and CLO = 3 (white), CHI = -1 (transparent). This gives a mountaintop that is completely white at the top, but lets a variable amount of the former color show through at the lower altitudes.

On line 1140 the lowlands around the mountain are painted first with a TYPE = 3 checkerboard blend and a DIR = 0 horizontal sweep, with the blended colors CLO = 3 (white) and CHI = 2 (green). Then this same area is repainted in line 1150 with TYPE = 2 pixel painting, DIR = 0 horizontal sweep, FAC = .5 for a 50-50 even mix of CLO and CHI, where CLO = -1 (transparent) and CHI = 2 (green).

The foreground scene is first painted on line 1190 with TYPE = 3 checkerboard blending, with a DIR = 0 horizontal sweep technique, and the two blended colors CLO = 2 (green) and CHI = 0 (blue). Then the foreground is repainted with TYPE = 2 pixel painting with DIR = 0 horizontal sweep, FAC = 3 color blending, with the CLO color at the top predominating over the CHI color at the bottom. The CLO = -1 transparent color will allow the previous blue-green blend to show through, while the CHI = 2 green color gives some texture overlay effects.

The river is defined at line 1230, first as a TYPE = 3 checkerboard blend with a horizontal sweep DIR = 0, and the two blended colors CLO = 0, the blue background, and CHI = 3, white. Then the river is repainted with TYPE = 1 bar painting, using DIR = 0 horizontal bars, SPA = 3 spaces between bars, FAC = 1 for the spaces to increase across the polygon, and CLO = 3 for the white color.

The next two DATA statements define the tops of two trees and, while the shapes and locations differ, they are both painted in the same way. They use TYPE = 2 pixel painting with a horizontal sweep DIR = 0 and a FAC = .2 for an even mixture of 80% CLO and 20% CHI, where CLO = 2 (green) and CHI = 0 (blue).

The tree trunks are both painted in the TYPE = 1 bar mode, using DIR = 0 horizontal bars with SPA = 0 for no spaces between the bars, and FAC = 0 for no variability in the bar spacing across the polygon. The color register CLO = 1 gives a reddish maroon shade.

The remaining details of the picture consist of two human figures. These figures are so small that the PolyPaint routine cannot be used efficiently to display them. The high-resolution detail required by figures like these must be achieved by specific PLOT and DRAWTO commands. The PolyPaint routine is suitable for painting relatively large screen areas.

At this point you can take this Picture program and change the Graphics mode, SETCOLOR assignments, and DATA statements and use it to paint your own picture. However, the key word here is *paint,* not *create.* We don't create our picture with this program, we just manifest it.

The actual creation of the picture is done away from the computer, with a pencil and a paper that has a grid pattern marked off corresponding to the screen's horizontal and vertical coordinate numbers for the Graphics mode that we choose. We then sketch out our picture on this grid paper, noting the various areas, their vertex point values, the colors we would like to assign, and the painting and overlay methods that we might want to use. We also must define which high-resolution detail we must draw pixel-by-pixel or line-by-line. Only after all this homework is done can we sit down at our machine and paint our picture.

## The Palette Utility

I found that, to use this PolyPaint subroutine effectively, I needed a utility program which allowed me to study the effects available by using the various painting command options, a variety of overlay effects, different color and luminance values, and the different resolutions available from the various Graphics modes. The utility program that allows this study is Program 3, the Palette program.

A brief survey of this program will prove informative. The first question asked by the program is for the Graphics mode. The program presently allows the BASIC modes 3 through 8. Depending upon the mode selected, two scale factor variables, E and F, are assigned appropriate numeric values. As we scan through the remainder of this program we see that every X, Y value in a PLOT or DRAWTO statement is multiplied by these scale factors.

The initial color assignment to the four registers is done between lines 370 and 430. The background is white and the color registers 1, 2, and 3 are assigned the colors red, green, and blue.

If you RUN this program you will see that it proceeds to display three rectangular areas showing the colors in registers in 1, 2, and 3; the areas are numbered appropriately. Above each of these primary colors is a checkerboard blend of the primary color with the background color. Below the primary colors are three rectangular areas that show checkerboard blends of each of the three primary colors against each other. The connecting lines clarify which color is being blended with which.

Then, on the right part of the picture, four areas are reserved for us to try out various patterns. On line 1480 we are asked which area we wish to use. We can answer this question with the value zero to four. If we answer with the value zero, then we are given the option of redefining any of the hue or luminance values in the four color registers. We are shown the hue and luminance numbers for each register, and are then asked which register number we wish to change.

After we select a register number we then use joystick zero to change the hue and luminance for that register. Moving the stick sideways changes the hue and moving it forward or back changes the luminance. When we are ready to "fix" that hue and luminance we press the trigger button. We then return to the "Which COLOR" question. At this point we can return to the "Which AREA" question by RETURNing with the value 99.

It should be noted that if, in response to any question, we return with the value 99, then we will always be shifted back to the "Which AREA" question in the program. Thus, if we change our minds while specifying a certain type of painting technique, we can always abort the sequence by typing the number 99.

Instead of giving suggestions on how this Palette program might be used, I'll just let you explore it yourself. It is self-explanatory anyway, and fairly well error-protected.

It should be understood that the area of textured graphics is completely open-ended. The texturizing options in this version of PolyPaint just scratch the surface, so to speak, of what is possible within this category. Anyone who wants to can modify or expand this program to incorporate a much greater variety of effects. If you develop something in this area, I would certainly be happy to learn about it.

The Palette program also can be expanded. The present version of this program only allows the BASIC graphics modes 3 to 8. As we learn more about the Atari system, we should be able to modify this program to include the additional ANTIC graphics modes and thus expand our repertoire of picture-creating capabilities.

**PROGRAM 1. Textured Graphics.**

```
17000 REM ===========================
17005 REM POLYPAINT Subroutine
17010 REM Polygon Color Painting
17015 REM ...........................
17020 REM by Phil Dunn, with The ECK*
17025 REM ...........................
17030 REM Enter with the values...
17035 REM NP = No. of vertex points.
17040 REM X()= DIM array of X values
17045 REM Y()= DIM array of Y values
17050 REM S()= DIM array used here
17055 REM T()= DIM array used here
17060 REM TYPE= Type of painting
17065 REM "{3 SPACES}= 1 for bar painting
17070 REM "{3 SPACES}= 2 for pixel painting
17075 REM "{3 SPACES}= 3 for checkerboard
17080 REM "{3 SPACES}= 4 to draw a line
17085 REM ...........................
17090 REM ****** Bar painting input:
17095 REM DIR= Bar Direction
17100 REM "  = 0 for horizontal bars
17105 REM "  = 1 for vertical bars
17110 REM SPA= Spaces between bars
17115 REM "  = 0 for no spaces
17120 REM "  >=1 to skip spaces
17125 REM FAC= Variable space factor
17130 REM " = 1 for increasing spaces
17135 REM " = 0 for constant spaces
17140 REM " =-1 for decreasing spaces
17145 REM CLO= Color register number
17150 REM ...........................
17155 REM ***** Pixel painting input:
17160 REM DIR= Sweep Direction
17165 REM "  = 0 for horizontal sweep
17170 REM "  = 1 for vertical sweep
17175 REM FAC= Blending factor from
17180 REM .{5 SPACES}low end to high end
17185 REM "  =0-.99 for an even mix
17190 REM "{5 SPACES}<.5=more CLO color
17195 REM "{5 SPACES}>.5=More CHI color
17200 REM "  >=1 for uneven color mix
17205 REM "{5 SPACES}=1-3 for more low end
17210 REM "{5 SPACES}>7 for more high end
17215 REM CLO= Low end color reg.
17220 REM CHI= High end color reg.
17225 REM Note: Setting CLO or CHI to
```

```
17230 REM -1 will hold the previous
17235 REM color on those pixels.
17240 REM ..........................
17245 REM **** Checkerboard painting:
17246 REM DIR= Sweep Direction
17247 REM "   = 0 for horizontal sweep
17248 REM "   = 1 for vertical sweep
17250 REM CLO= First color register
17255 REM CHI= Second color register
17260 REM ..........................
17265 REM *********** To Draw A Line:
17270 REM X(0),Y(0)=Start point
17275 REM X(1),Y(1)=End point
17280 REM CLO=Start color
17285 REM CHI=Alternate color
17290 REM ..........................
17295 REM Uses the variable names Q0,
17300 REM Q1,Q2,Q3,Q4,Q5,COL,IM,IP,DL
17305 REM K,L,M,N,R,MAX,MIN,MXMN,NOW
17310 REM ..........................
17315 Q0=0:Q1=1:Q2=1000:IF TYPE=4 THEN 17660
17320 FOR M=Q1 TO NP:N=M+Q1:IF N>NP THEN N=Q
      1
17325 IF X(M)=X(N) THEN S(M)=Q2:GOTO 17340
17330 REM Slopes=S(), Intercepts=T()
17335 S(M)=(Y(N)-Y(M))/(X(N)-X(M)):T(M)=Y(M)
      -S(M)*X(M)
17340 NEXT M:MAX=-Q2:MIN=Q2
17345 FOR M=Q1 TO NP:Q3=Y(M):IF DIR>Q0 THEN
      Q3=X(M)
17350 IF MAX<Q3 THEN MAX=Q3
17355 IF MIN>Q3 THEN MIN=Q3:N=M
17360 NEXT M:MXMN=MAX-MIN:NOW=MIN:IM=N-Q1:IF
       IM<Q1 THEN IM=NP
17365 IP=N+Q1:IF IP>NP THEN IP=Q1
17375 M=17395:IF DIR>Q0 THEN M=17420
17380 IF TYPE=Q1 THEN COLOR CLO
17385 GOTO 17615
17390 REM Horizontal..................
17395 IF S(N)=Q2 OR S(N)=Q0 THEN Q3=X(N):GOT
      O 17405
17400 Q3=(NOW-T(N))/S(N)
17405 IF S(IM)=Q2 OR S(IM)=Q0 THEN Q4=X(IM):
      GOTO 17445
17410 Q4=(NOW-T(IM))/S(IM):GOTO 17445
17415 REM Vertical....................
17420 IF S(N)=Q2 THEN Q3=Y(N):GOTO 17430
17425 Q3=NOW*S(N)+T(N)
17430 IF S(IM)=Q2 THEN Q4=Y(IM):GOTO 17445
17435 Q4=NOW*S(IM)+T(IM)
```

```
17440 REM ..........................
17445 IF TYPE>Q1 THEN 17480
17450 REM BAR-FILL..................
17455 IF DIR=Q0 THEN PLOT Q3,NOW:DRAWTO Q4,N
      OW
17460 IF DIR>Q0 THEN PLOT NOW,Q3:DRAWTO NOW,
      Q4
17465 Q3=(NOW-MIN)/MXMN:Q4=INT(SPA*(2*FAC*Q3
      +1-FAC))
17470 NOW=NOW+Q1+Q4:GOTO 17615
17475 REM PIXEL-FILL (PF)...........
17480 Q5=INT(ABS(Q4-Q3)):DL=SGN(Q4-Q3):L=Q3:
      R=FAC*(NOW-MIN)/MXMN
17485 IF TYPE<>3 THEN 17520
17490 REM CHECKERBOARD .............
17495 Q3=CLO:Q4=CHI:R=INT(L+NOW+0.5):IF R=2*
      INT(R/2) THEN Q3=CHI:Q4=CLO
17500 R=DL+DL+0.5
17505 IF DIR=0 THEN FOR K=Q1 TO Q5/2:COLOR Q
      3:PLOT L,NOW:COLOR Q4:PLOT L+DL,NOW:L=
      INT(L+R):NEXT K
17506 IF DIR=1 THEN FOR K=Q1 TO Q5/2:COLOR Q
      3:PLOT NOW,L:COLOR Q4:PLOT NOW,L+DL:L=
      INT(L+R):NEXT K
17510 IF Q5=2*INT(Q5/2) THEN 17600
17511 IF DIR=0 THEN COLOR Q3:PLOT L,NOW
17512 IF DIR=1 THEN COLOR Q3:PLOT NOW,L
17515 GOTO 17600
17520 IF FAC<Q1 AND DIR=Q0 THEN 17540
17525 IF FAC<Q1 AND DIR>Q0 THEN 17555
17530 IF FAC>=Q1 AND DIR=Q0 THEN 17570
17535 IF FAC>=Q1 AND DIR>Q0 THEN 17585
17540 FOR K=Q0 TO Q5:COL=CLO:IF RND(Q0)<FAC
      THEN COL=CHI
17545 IF COL>=Q0 THEN COLOR COL:PLOT L,NOW
17550 L=L+DL:NEXT K:GOTO 17600
17555 FOR K=Q0 TO Q5:COL=CLO:IF RND(Q0)<FAC
      THEN COL=CHI
17560 IF COL>=Q0 THEN COLOR COL:PLOT NOW,L
17565 L=L+DL:NEXT K:GOTO 17600
17570 FOR K=Q0 TO Q5:COL=CLO:IF R*RND(Q0)>0.
      5 THEN COL=CHI
17575 IF COL>=Q0 THEN COLOR COL:PLOT L,NOW
17580 L=L+DL:NEXT K:GOTO 17600
17585 FOR K=Q0 TO Q5:COL=CLO:IF R*RND(Q0)>0.
      5 THEN COL=CHI
17590 IF COL>=Q0 THEN COLOR COL:PLOT NOW,L
17595 L=L+DL:NEXT K:GOTO 17600
17600 NOW=INT(NOW+Q1)
17605 REM Check for vertex point
```

```
17610 REM passover & end of polygon
17615 Q3=Y(IP):Q4=Y(IM):IF DIR>Q0 THEN Q3=X(
      IP):Q4=X(IM)
17620 IF NOW<=Q3 THEN 17635
17625 IF Q3=MAX THEN RETURN
17630 N=IP:IP=IP+Q1:IF IP>NP THEN IP=Q1
17635 IF NOW<=Q4 THEN GOTO M
17640 IF Q4=MAX THEN RETURN
17645 IM=IM-Q1:IF IM<Q1 THEN IM=NP
17650 GOTO M
17655 REM Line Drawing...............
17660 K=X(Q1)-X(Q0):Q2=ABS(K)
17665 L=Y(Q1)-Y(Q0):Q3=ABS(L)
17670 IF Q2>=Q3 THEN N=Q2:Q4=SGN(K):Q5=L/Q2
17675 IF Q2<Q3 THEN N=Q3:Q5=SGN(L):Q4=K/Q3
17680 Q2=X(Q0):Q3=Y(Q0):S(Q0)=CLO:S(Q1)=CHI:
      IP=Q0
17685 FOR M=Q1 TO N:COLOR S(IP):PLOT Q2,Q3:Q
      2=Q2+Q4:Q3=Q3+Q5:IP=Q1-IP:NEXT M
17690 RETURN
17695 REM ===========================
```

**PROGRAM 2. Textured Graphics.**

```
100 REM =============================
105 REM ={5 SPACES}PICTURE Program
    {7 SPACES}=
107 REM ={8 SPACES}for the{12 SPACES}=
110 REM ={3 SPACES}POLYPAINT Subroutine
    {4 SPACES}=
120 REM =============================
125 REM ={11 SPACES}by:{13 SPACES}=
130 REM ={8 SPACES}Phil Dunn{10 SPACES}=
140 REM ={6 SPACES}12 Monroe Ave.{7 SPACES}=
150 REM ={3 SPACES}Hicksville, NY 11801
    {4 SPACES}=
160 REM =============================
220 REM Set the graphics mode
230 REM and the color registers...
240 GRAPHICS 7+16
250 SETCOLOR 0,5,6:REM C.1 = MAROON
260 SETCOLOR 1,10,8:REM C.2 = GREEN
270 SETCOLOR 2,9,10:REM C.3 = WHITE, WINDO
280 SETCOLOR 4,8,8:REM C.0 = BLUE, BACKG.
290 REM =============================
300 DIM X(15),Y(15),S(15),T(15)
310 POLYPAINT=17315
330 REM ============================
340 REM Read areas and fill them in
350 REM ............................
360 REM First read the TYPE of painting to b
    e done for this area.
370 A=0
380 READ TYPE
390 A=A+1
400 REM Test for end of data
410 IF TYPE=999 THEN 1020
420 REM Now go to proper line to read the re
    st of the data for this TYPE.
430 IF TYPE=4 THEN 820
440 IF TYPE=3 THEN 730
450 IF TYPE=2 THEN 620
460 IF TYPE<>1 THEN ? "TYPE=";TYPE,"for AREA
    =";A:STOP
470 REM ............................
480 REM Read data for TYPE 1
490 REM bar painting.
500 READ DIR
510 IF DIR<>0 AND DIR<>1 THEN ? "DIR=";DIR,"
     for AREA=";A:STOP
520 READ SPA
```

```
530 IF SPA<0 THEN ? "SPA=";SPA,"for AREA=";A
    :STOP
540 READ FAC
550 IF FAC<-1 OR FAC>1 THEN ? "FAC=";FAC,"fo
    r AREA=";A:STOP
560 READ CLO
570 IF CLO<-1 OR CLO>3 THEN ? "CLO=";CLO,"fo
    r AREA=";A:STOP
580 GOTO 920
590 REM ............................
600 REM Read data for TYPE 2
610 REM pixel painting.
620 READ DIR
630 IF DIR<>0 AND DIR<>1 THEN ? "DIR=";DIR,"
    for AREA=";A:STOP
640 READ FAC
650 IF FAC<0 THEN ? "FAC=";FAC,"for AREA=";A
    :STOP
660 READ CLO
670 IF CLO<-1 OR CLO>3 THEN ? "CLO=";CLO,"fo
    r AREA=";A:STOP
680 READ CHI
690 IF CHI<-1 OR CHI>3 THEN ? "CHI=";CHI,"fo
    r AREA=";A:STOP
700 GOTO 920
710 REM ............................
720 REM Input for TYPE 3
725 REM checkerboard painting.
730 READ DIR
735 IF DIR<>0 AND DIR<>1 THEN ? "DIR=";DIR,"
    for AREA=";A:STOP
740 READ CLO
750 IF CLO<0 OR CLO>3 THEN ? "CLO=";CLO,"for
     AREA=";A:STOP
760 READ CHI
770 IF CHI<0 OR CHI>3 THEN ? "CHI=";CHI,"for
     AREA=";A:STOP
780 GOTO 920
790 REM ............................
800 REM Input for TYPE 4,
810 REM to draw a line.
820 READ CLO
830 IF CLO<0 OR CLO>3 THEN ? "CLO=";CLO,"for
     AREA=";A:STOP
840 READ CHI
850 IF CHI<0 OR CHI>3 THEN ? "CHI=";CHI,"for
     AREA=";A:STOP
860 READ X,Y:X(0)=X:Y(0)=Y
870 READ X,Y:X(1)=X:Y(1)=Y
880 GOSUB POLYPAINT
```

```
890 GOTO 380
900 REM ..............................
910 REM Now read the polygon perimeter data
920 READ NP:REM Number of points
930 IF NP<3 THEN ? "NP=";NP,"for AREA=";A:ST
    OP
940 FOR N=1 TO NP
950 READ X,Y
960 X(N)=X
970 Y(N)=Y
980 NEXT N
990 REM Now let the subroutine fill it in
1000 GOSUB POLYPAINT
1010 GOTO 380
1020 REM ============================
1030 REM ={3 SPACES}Scene from the book
     {4 SPACES}=
1040 REM = 'Stranger By The River'  =
1050 REM ={4 SPACES}by Paul Twitchell
     {5 SPACES}=
1060 REM ============================
1070 REM MOUNTAIN
1080 DATA 3,1,3,2,6,54,42,68,30,85,22,110,35
     ,118,45,54,45
1090 DATA 2,1,9,2,-1,5,54,42,68,30,85,22,85,
     45,54,45
1100 DATA 2,1,2,-1,2,4,85,22,110,35,118,45,8
     5,45
1110 REM MOUNTAIN TOP
1120 DATA 2,0,1,3,-1,6,85,22,98,29,95,30,88,
     32,78,29,74,27
1130 REM LOWLANDS
1140 DATA 3,0,3,2,3,0,45,80,40,159,45
1150 DATA 2,0,.5,-1,2,3,0,45,80,40,159,45
1180 REM FOREGROUND
1190 DATA 3,0,2,0,4,0,69,159,66,159,95,0,95
1200 DATA 2,0,3,-1,2,4,0,80,159,80,159,95,0,
     95
1210 REM RIVER
1230 DATA 3,0,0,3,12,0,50,159,50,159,66,140,
     68,125,70,105,74,80,77,60,78,47,78,40,7
     7,25,75,0,69
1260 DATA 1,0,3,1,3,6,0,45,159,45,159,60,90,
     75,40,75,0,60
1270 REM LEFT TREE TOP
1280 DATA 2,0,.2,2,0,12,25,25,37,34,33,36,36
     ,40,31,43,33,55,18,59,5,50,13,42,8,37,1
     6,34,12,30
1310 REM RIGHT TREE TOP
1320 DATA 2,0,.2,2,0,8,130,32,149,39,145,46,
     153,56,135,68,116,61,122,52,115,47
1340 REM LEFT TREE TRUNK
1350 DATA 1,0,0,0,1,6,11,80,16,56,19,52,21,5
     7,17,81,14,83
1360 REM RIGHT TREE TRUNK
1370 DATA 1,0,0,0,1,6,131,83,130,65,133,59,1
     36,65,141,82,136,85
1380 REM END OF DATA
1390 DATA 999
1400 REM ============================
1410 REM Human figures...
1420 REM Paul
1430 COLOR 3
1440 XX=55
1450 YY=80
1460 PLOT XX-1,YY+3:DRAWTO XX-1,YY+8
1470 PLOT XX,YY:DRAWTO XX,YY+8
1480 PLOT XX+1,YY+3:DRAWTO XX+1,YY+8
1490 PLOT XX+4,YY+4:DRAWTO XX+4,YY+8
1500 PLOT XX+1,YY:PLOT XX+1,YY+1
1510 PLOT XX+2,YY+4:PLOT XX+3,YY+4
1520 PLOT XX+3,YY+5:PLOT XX+3,YY+6
1530 PLOT XX+2,YY+6:PLOT XX+2,YY+7
1540 PLOT XX+5,YY+8
1550 PLOT XX+1,YY:PLOT XX+1,YY+1
1560 PLOT XX+2,YY+4:PLOT XX+3,YY+4
1570 REM Rebezar
1580 COLOR 1
1590 XX=65
1600 YY=70
1610 PLOT XX-1,YY+2:DRAWTO XX-1,YY+13
1620 PLOT XX,YY:DRAWTO XX,YY+10
1630 PLOT XX+1,YY:DRAWTO XX+1,YY+10
1640 PLOT XX+2,YY+2:DRAWTO XX+2,YY+13
1650 PLOT XX+5,YY+1:DRAWTO XX+5,YY+13
1660 PLOT XX-4,YY+1:PLOT XX-3,YY+2
1670 PLOT XX-2,YY+2:PLOT XX+3,YY+3
1680 PLOT XX+4,YY+4
1690 GOTO 1690
1700 REM ============================
```

**PROGRAM 3. Textured Graphics.**

```
100 REM ==============================
105 REM ={5 SPACES}PALETTE Program
    {7 SPACES}=
110 REM =A Color-Texture Development=
120 REM =  Utility Program For The   =
130 REM ={3 SPACES}POLYPAINT Subroutine
    {4 SPACES}=
140 REM ==============================
145 REM ={11 SPACES}by:{13 SPACES}=
150 REM ={8 SPACES}Phil Dunn{10 SPACES}=
160 REM ={6 SPACES}12 Monroe Ave.{7 SPACES}=
170 REM ={3 SPACES}Hicksville, NY 11801
    {4 SPACES}=
180 REM ==============================
240 DIM A$(2),X(4),Y(4),S(4),T(4),U(50)
245 DIM HU(3),LU(3)
250 POLYPAINT=17315
260 REM ==============================
270 REM Input mode & scale palette
280 GRAPHICS 0
290 ? "Which Graphics MODE (3-8)";
300 TRAP 290:INPUT MODE
310 IF MODE<3 OR MODE>8 THEN 290
320 GRAPHICS MODE
325 POKE 752,1:REM Blank Cursor
330 IF MODE=3 THEN E=0.25:F=0.25
340 IF MODE=4 OR MODE=5 THEN E=0.5:F=0.5
350 IF MODE=6 OR MODE=7 THEN E=1:F=1
360 IF MODE=8 THEN E=2:F=2
370 REM ==============================
372 REM Initial color assignment...
374 HU(0)=0:LU(0)=8:REM =WHITE
376 HU(1)=5:LU(1)=8:REM =RED
378 HU(2)=10:LU(2)=6:REM =GREEN
380 HU(3)=8:LU(3)=8:REM =BLUE
390 SETCOLOR 0,HU(1),LU(1)
400 SETCOLOR 1,HU(2),LU(2)
410 SETCOLOR 2,HU(3),LU(3):REM WINDOW
420 SETCOLOR 4,HU(0),LU(0):REM BACKGR
430 REM ==============================
440 REM Primary colors numbers
450 COLOR 1
460 REM Number 1...
470 PLOT 3*E,22*F
480 DRAWTO 3*E,28*F
490 REM Number 2...
500 PLOT 19*E,22*F
510 DRAWTO 21*E,22*F
```

```
520 PLOT 22*E,23*F
530 DRAWTO 19*E,28*F
540 DRAWTO 22*E,28*F
550 REM Nunber 3...
560 PLOT 37*E,22*F
570 DRAWTO 40*E,22*F
580 DRAWTO 40*E,28*F
590 DRAWTO 37*E,28*F
600 PLOT 38*E,25*F
610 DRAWTO 40*E,25*F
620 REM ==============================
630 REM Display Primary Colors
640 NP=4:TYPE=1:DIR=0:SPA=0:FAC=0
650 FOR I=1 TO 3
660 CLO=I
670 FOR J=1 TO 4
680 READ X,Y:X(J)=X*E:Y(J)=Y*F
690 NEXT J
700 GOSUB POLYPAINT
710 NEXT I
720 REM ==============================
730 REM Display Secondary Colors
740 TYPE=3:REM Checkerboard
745 DIR=0
750 FOR I=1 TO 6
760 FOR J=1 TO 4
770 READ X,Y:X(J)=X*E:Y(J)=Y*F
780 NEXT J
790 IF I<=3 THEN CLO=0:CHI=I
800 IF I=4 THEN CLO=1:CHI=2
810 IF I=5 THEN CLO=1:CHI=3
820 IF I=6 THEN CLO=2:CHI=3
830 GOSUB POLYPAINT
840 NEXT I
850 REM =============================
860 REM Areas for primary colors....
870 DATA 0,45,0,30,15,30,15,45
880 DATA 18,45,18,30,33,30,33,45
890 DATA 36,45,36,30,51,30,51,45
900 REM Areas for secondary colors..
910 DATA 0,20,0,5,15,5,15,20
920 DATA 18,20,18,5,33,5,33,20
930 DATA 36,20,36,5,51,5,51,20
940 DATA 0,70,0,55,15,55,15,70
950 DATA 18,70,18,55,33,55,33,70
960 DATA 36,70,36,55,51,55,51,70
970 REM Connect the colors
980 DATA 8,30,8,20,8,45,8,55,12,45,22,55
990 DATA 26,30,26,20,22,45,12,55,28,45,39,55
1000 DATA 44,30,44,20,44,45,44,55,40,45,29,55
```

```
1010 REM Palette Display Areas.......
1020 DATA 60,35,60,0,105,0,105,35
1030 DATA 115,35,115,0,159,0,159,35
1040 DATA 60,75,60,40,105,40,105,75
1050 DATA 115,75,115,40,159,40,159,75
1060 REM ============================
1070 REM Connect the colors
1080 FOR I=1 TO 3
1090 COLOR I
1100 FOR J=1 TO 3
1110 READ X1,Y1,X2,Y2
1120 PLOT X1*E,Y1*F
1130 DRAWTO X2*E,Y2*F
1140 NEXT J:NEXT I
1150 REM ============================
1160 COLOR 1
1170 REM Number the palette areas...
1180 REM Number 1
1190 PLOT 57*E,15*F
1200 DRAWTO 57*E,25*F
1210 REM Number 2
1220 PLOT 109*E,15*F
1230 DRAWTO 112*E,15*F
1240 DRAWTO 113*E,17*F
1250 DRAWTO 109*E,25*F
1260 DRAWTO 113*E,25*F
1270 REM Number 3
1280 PLOT 54*E,55*F
1290 DRAWTO 58*E,55*F
1300 DRAWTO 58*E,65*F
1310 DRAWTO 54*E,65*F
1320 PLOT 55*E,60*F
1330 DRAWTO 58*E,60*F
1340 REM Number 4
1350 PLOT 110*E,57*F
1360 DRAWTO 110*E,60*F
1370 DRAWTO 113*E,60*F
1380 PLOT 113*E,55*F
1390 DRAWTO 113*E,65*F
1400 NP=4
1410 REM ============================
1420 REM = NOTE:  TO RESTART THE{4 SPACES}=
1430 REM = INPUT SEQUENCE AT ANY{4 SPACES}=
1440 REM = TIME BACK TO THE 'AREA'  =
1450 REM = INPUT REQUEST, RETURN{4 SPACES}=
1460 REM = WITH THE VALUE 99.{7 SPACES}=
1470 REM ============================
1480 ? "Which AREA to use (0-4)";
1490 TRAP 1480:INPUT A
1500 IF A<0 OR A>4 THEN 1480
```

```
1505 IF A=0 THEN 2140
1510 RESTORE 1010+A*10
1520 FOR I=1 TO 4
1530 READ X,Y:X(I)=INT(X*E):Y(I)=INT(Y*F)
1540 NEXT I
1550 REM ============================
1560 ? "What painting TYPE (1, 2, or 3),"
1570 ? "Bar, pixel, or checkerboard";
1580 TRAP 1560:INPUT TYPE
1590 IF TYPE=99 THEN 1480
1600 IF TYPE<1 OR TYPE>3 THEN 1560
1610 IF TYPE=2 THEN 1850
1620 IF TYPE=3 THEN 2020
1630 REM ============================
1640 REM Bar painting input...
1650 ? "HORIZ. or VERTICAL Bars (0 or 1)";
1660 TRAP 1650:INPUT DIR
1670 IF DIR=99 THEN 1480
1680 IF DIR<>0 AND DIR<>1 THEN 1650
1690 ? "SPACES between bars, >=0  ";
1700 TRAP 1690:INPUT SPA
1710 IF SPA=99 THEN 1480
1720 IF SPA<0 THEN 1690
1730 ? "Space VARIABILITY (-1 to +1)";
1740 TRAP 1730:INPUT FAC
1750 IF FAC=99 THEN 1480
1760 IF FAC<-1 OR FAC>1 THEN 1730
1770 ? "COLOR register (0-3)";
1780 TRAP 1770:INPUT CLO
1790 IF CLO=99 THEN 1480
1800 IF CLO<0 OR CLO>3 THEN 1770
1810 GOSUB POLYPAINT
1820 GOTO 1480
1830 REM ============================
1840 REM PIXEL Painting...
1850 ? "HORIZ. or VERTICAL Sweep (0 or 1)";
1860 TRAP 1850:INPUT DIR
1870 IF DIR=99 THEN 1480
1880 IF DIR<>0 AND DIR<>1 THEN 1850
1890 ? "BLENDING Factor, 0-0.99, or >=1";
1900 TRAP 1890:INPUT FAC
1910 IF FAC=99 THEN 1480
1920 IF FAC<0 THEN 1890
1930 ? "LOW Color (-1 to 3)";
1940 TRAP 1930:INPUT CLO
1950 IF CLO=99 THEN 1480
1960 ? "HIGH Color (-1 to 3)";
1970 TRAP 1960:INPUT CHI
1980 IF CHI=99 THEN 1480
1990 GOSUB POLYPAINT
```

```
2000 GOTO 1480
2010 REM ============================
2015 REM CHECKERBOARD Painting...
2020 ? "HORIZ. or VERTICAL Sweep (0 or 1)";
2024 TRAP 2020:INPUT DIR
2026 IF DIR=99 THEN 1480
2028 IF DIR<>0 AND DIR<>1 THEN 2020
2030 ? "First Color (0 to 3)";
2040 TRAP 2030:INPUT CLO
2050 IF CLO=99 THEN 1480
2055 IF CLO<0 OR CLO>3 THEN 2030
2060 ? "Alternate Color (0 to 3)";
2070 TRAP 2060:INPUT CHI
2080 IF CHI=99 THEN 1480
2085 IF CHI<0 OR CHI>3 THEN 2060
2090 GOSUB POLYPAINT
2100 GOTO 1480
2110 REM ============================
2130 REM Redefine the colors...
2140 ? "Reg.No. = Hue, Luminance"
2145 ? "0=";HU(0);",";LU(0),"1=";HU(1);",";L
     U(1),"2=";HU(2);",";LU(2),"3=";HU(3);",
     ";LU(3)
2150 ? "Which COLOR Register (0-3) ";
2160 TRAP 2140:INPUT I
2170 IF I=99 THEN 1480
2180 IF I<0 OR I>3 THEN 2140
2190 K=HU(I):L=LU(I):GOTO 2230
2200 IF STRIG(0)=0 THEN 2140
2210 IF HU(I)=INT(K) AND LU(I)=2*INT(L/2) TH
     EN 2260
2220 HU(I)=INT(K):LU(I)=2*INT(L/2)
2230 ? "Reg. ";I;" = Hue ";HU(I);" , Luminan
     ce ";LU(I)
2240 J=I-1:IF J<0 THEN J=4
2250 SETCOLOR J,HU(I),LU(I)
2255 HU(I)=INT(K):LU(I)=2*INT(L/2)
2260 IF STICK(0)=7 THEN K=K+0.1:IF K>16 THEN
      K=0
2270 IF STICK(0)=11 THEN K=K-0.1:IF K<0 THEN
      K=15
2280 IF STICK(0)=14 THEN L=L+0.2:IF L>16 THE
     N L=0
2290 IF STICK(0)=13 THEN L=L-0.2:IF L<0 THEN
      L=14
2292 REM Joystick 0 controls:
2293 REM Left-Right changes hue
2294 REM Foward-Back changes luminance
2295 REM Press Trigger to fix selection
2300 GOTO 2200
```

# Part 3: Multi-Colored Graphics In Mode 8

Phil Dunn

*Into the world of Graphics mode 8, with multi-colored displays. This tutorial and commentary covers both the CTIA and the GTIA chips. You'll find the explanations and utility programs invaluable. 32K RAM memory is recommended, but, with judicious cutting, you can run this with 24K.*

Graphics Mode 8 provides the highest resolution images that we can generate with the Atari system, with a horizontal grid of 320 and a vertical grid of 192 (or 160 with a split screen). Therefore, a mastery of the mysteries of Mode 8 will enable us to develop some of the highest quality images that the Atari system can provide. This article offers you that ability, with some user-friendly programs that make it as easy as apple pie. If you want to, you can generate your own version of the Atari Video Easel, or you can use the programs to make really fine pictures.

These routines will require 32K of memory to work in Mode 8 graphics. With all REM statements removed, and a reduction in some nonessential functions, these routines should work with only 24K of memory.

There is a rumor that Mode 8 graphics allows only one color. This rumor states that the SETCOLOR 2 command determines the hue and luminance of the background, and the SETCOLOR 1 command determines just the luminance of a point or line drawn with the PLOT-DRAWTO commands. The only other SETCOLOR command that has any effect in Mode 8 is SETCOLOR 4, which only controls the hue and luminance of the border.

This rumor has an impeccable source: the Atari BASIC Reference Manual. Let's check it out. We go over to our favorite machine and flip on the power switches for our computer and TV monitor, and see our friendly 'READY' message pop up at the top of the screen.

From past encounters of this kind we know that we are in Mode 0. So, we type in the direct command, GRAPHICS 8. The screen flips and now we see the READY at the bottom of the screen. "Aha!"

we think, "Mode 8 with our split-screen text window at the bottom."

Let's draw a line. We type in the command, COLOR 1:PLOT 5,80:DRAWTO 315,80 and a horizontal line appears. Now we can play with SETCOLOR commands to our heart's content, and we will only conclude that the manual is telling the truth, the whole truth, and nothing but the truth, so help them Atari!

Now let's run a different experiment. As in any good experiment, we must carefully set the conditions. With SETCOLOR 2,0,0 we set a black background, and with SETCOLOR 1,0,12 we get set for high-contrast lines. Now, with the command PLOT 160,150:DRAWTO 160,5 we see a brown line if we have the CTIA chip, or we see a blue line if we have the GTIA chip.

But you may not see brown or blue. The color you see, and all the colors that are mentioned in the remainder of this article, are dependent upon the condition and alignment of your system. While the colors you see may be different than the ones described here, the principles and techniques for obtaining the variety of colors remain the same.

Now let's enter the command PLOT 101,150: DRAWTO 101,5, and we will see a blue vertical line if we have the CTIA chip, or a brown line with the GTIA. Why are the two lines different colors? It's just dependent upon whether the vertical line has an even or an odd value for its X coordinate. All vertical lines with an even X value will be one color, and all with an odd X value will be the other. This is true for all PLOT points also. If we wish to draw a line at any angle by PLOTting only those line points at the odd or even X coordinate values, then we will obtain a blue or brown line.

Notice that I am saying that the color we obtain (as a function of whether the X coordinate parity is odd or even) is reversed between the CTIA and the GTIA chips. Both chips give us exactly the same variety of colors and textures in Mode 8. It is just the effect of the X coordinate parity that is reversed. As far as I know, this type of color difference between the two chips does not exist in any mode other than Graphics 8.

In the remainder of this article, when I refer to colors, I will first indicate the color we obtain with the CTIA chip, and then, alongside it and in parentheses, the color obtained with the GTIA chip.

If we draw another line with the command PLOT 160,80:DRAWTO 240,120 we see a green (red) line. Typing PLOT 101,80:DRAWTO 181,120 gives a red (green) line. The rule here is that all sloping lines with an X/Y ratio slope of 2/1 will be either green or red, depending upon the X coordinate of the start point.

The command PLOT 10,10:DRAWTO 150,150 gives us a grey line. All lines drawn with an X/Y ratio slope of 1/1 will be grey.

These are what we will call the six primary colors of Mode 8 graphics. White (horizontal lines, or several lines adjacent to each other), brown and blue (vertical lines), green and red (2/1 slope lines), and grey (1/1) slope lines. All other sloping lines tend to give peppermint-stripe mixtures of these primary colors.

Now, next to the blue (brown) vertical line at 101,150-101,5 let's PLOT-DRAWTO another line at 102,80-102,60. This gives us a green (red) line, but one with a different texture than we saw in the 2/1 ratio version. If we PLOT-DRAWTO another line at 100,60-100,40 we get a red (green) line, again with a different texture than we saw previously.

If we enter the PLOT-DRAWTO command 159,80-199,100, it merges with our green (red) sloping line to give a warm grey. The command 201,100-241,120 converts our green (red) line to a cool grey.

The colors and textures that we obtain by drawing multiple lines that interact with one another we can call the secondary colors and textures of Mode 8 graphics. How many are there? I don't know. Later on in this article you will see over 60 that I have found. I only stopped at that point to write this up so others could explore this also.

Well, what is going on here with that Atari Manual? Is Atari trying to hide something from us about Mode 8? What's the story?

The Atari Manual is not lying. It is hiding something, though. It is hiding something that might only be confusing to the programming novice who is still struggling to grasp the implications of the various commands in BASIC and the Atari graphics modes. Also, what value is a color you can only get by drawing a line at some specific angle? (Plenty of value! More about this later.)

The answer to what this is all about lies in the structure of our video tubes and the way they generate colors. The face of these tubes is covered with a series of horizontal "scan lines" that consist of a repeating series of blue, green and red phosphor dots. These phosphor dots only glow their color if the electron beam gun in the neck of the tube shoots them. These dots are so small that we don't see them as dots, but only as the composite color of many dots. The variety of colors that we see on TV, and with which Atari provides us in its hue-luminance SETCOLOR command, is obtained by controlling the electron beam intensity to each dot. It is the balancing of the brightness between the blue, green, and red dots that provides us with the full spectrum of colors.

This feature of Mode 8 graphics occurs because when we draw a vertical line, that line is so thin that it cannot cover all three colors of the screen. When the X coordinate of the vertical line is odd (even), it hits mostly the blue dots, and we get a blue line. When the X coordinate is even (odd), it hits both red and green dots, giving us a brown line. The resolution of Atari Mode 8 graphics is almost as fine as the resolution of the TV screen dot pattern! Since our lines may not touch all three color dots, the line color cannot be adjusted effectively by balancing the color-dot intensities as is done in the lower resolution graphics modes. (Horizontal lines always cover all three dot colors.)

Now we understand Mode 8, and we recognize that it has colors, but how can we use them? Well, we can always draw blue and brown lines, but that is not where the action is. The action is where the possibility exists for a multitude of colors, patterns, and textures. The action lies not in drawing lines, but in coloring areas. And what is our simplest, most useful method for coloring areas? If you have read the previous articles you know the answer. It is the polygon fill technique.

If we want to think of the PLOT-DRAW commands as being our Atari colored pencils, then the polygon fill technique is our Atari paintbrush.

At this point it is most appropriate to scan the programs presented here. Program 1, the POLY8 subroutine, is the essential tool that we will use to harness the graphics power of Mode 8. The POLY8 subroutine is supported by a secondary subroutine called LINEP, given here as Program 2. LINEP is based upon a machine code program. For those interested, the machine code assembly listing is presented as Program 3. My appreciation to Bill Wilkinson for that fantastically useful article on Atari I/O Graphics (**COMPUTE!**, February 1982, #21), and to Judy Bogart of Atari for her advice and direction regarding the CIO method.

Program 4 is the PALETT8 program, which allows us to investigate and discover the colors, patterns, and textures inherent in Mode 8. Although it has been set up to work in Modes 6 and 7 also, it was primarily designed to study Mode 8 graphics. Naturally, it essentially depends upon the POLY8 subroutine which must be appended to it.

Program 5 is the PICTUR8 program. This program allows us to define a complete picture just by entering values in data statements. This, too, is essentially dependent upon the POLY8 subroutine.

These programs have been written with an abundance of REM statements. Even with all the REM statements removed, they will require more than 16K of memory to run in Graphics Mode 8. With the REM statements removed they should run with 24K, or they can be left in if you have more.

The Atari BASIC Reference Manual tells us that we can reduce the memory requirements of our programs by defining constant numeric values in variables, when the constants are used in more than two or three places in our program. I have decided as a regular practice to set up the beginning of all my programs and general purpose subroutines with variables with names beginning with the letter "C" to represent the most commonly used numeric constants. This procedure has been implemented in these programs.

The key to the use of these programs is in the POLY8 subroutine, so let's take a look at Program 1. The initial REM statements summarize its capabilities, so we'll review them. The first variable this subroutine uses is TYPE, which can vary from 1 to 7, depending upon how this routine will be used. For TYPE values of 1 to 6 this routine also requires a value for the variable NP, and values for the DIMensioned array variables X(i), Y(i), where i varies from 1 to NP.

When TYPE=1, for "Bar Painting," a defined polygon area will be filled with lines (or "bars") in a way which we can specify. The polygon is defined by its perimeter points, the X, Y values stored in the X(), Y() DIMensioned array variables. NP specifies the number of perimeter points around the polygon.

When TYPE=2, for "Pixel Painting," the defined polygon will be filled on a pixel-by-pixel basis, according to our specified instructions. When TYPE=3 the defined polygon area will only be outlined, by drawing a series of lines around its perimeter.

For values of TYPE that are greater than 3, we are no longer dealing with a defined polygon area. Later on I will show how these options enable us to specify a huge amount of picture detail with a minimum amount of program coding.

When TYPE=4 we simply color in the pixels, connected or disconnected, that are specified in the X(i), Y(i) array variables, where i varies from 1 to NP.

When TYPE=5 we PLOT-DRAWTO a series of lines specified in the array variables X(i), Y(i) and X(i+1), Y(i+1) where i varies from 1 to NP. Note that NP always refers to the number of X, Y pairs to be used. Therefore, for one line set NP=2, for two lines set NP=4, for three line set NP=6, etc.

When TYPE=6 we DRAWTO from one point to another, starting at the initial cursor position to X(1), Y(1), then to X(2), Y(2), then to X(NP), Y(NP).

When TYPE=7 a color register is assigned by the COLOR command using the value stored in the RA variable.

For cases where TYPE = 1 or TYPE = 2, additional information is required. The slope at which the bars are to be drawn, or the pixels are to be swept, is specified in the RA variable in terms of the X/Y ratio. RA = 0 for vertical lines, + 1 or -1 for lines at + 45 degrees or 45 degrees, and RA = 100 for horizontal lines.

As we have previously seen, the factors that determine the line colors in Mode 8 are the line slope and the X-axis odd/even parity. The zero element of the P() array is used to specify the parity option. If P(0) = 0 then the parity option is bypassed. If P(0) = 1 then odd parity is selected, and if P(0) = 2 then the even parity option is chosen.

The remaining elements in the P(i) array, for values of i greater than zero, are used to specify the line spacing sequence, and determine whether or not the parity option is to be applied to that line. The parity option, if selected, always is applied to the first line drawn. All succeeding lines can have the parity option applied or not, at our specification. If the corresponding value of P(i) is negative, then the parity option will be applied to that line. If P(i) has a positive value, then the parity option will be bypassed for that line.

The magnitude of P(i) determines the increment to move to draw the next line. If the magnitude of P(i) is equal to 1, then the next line will be drawn immediately adjacent to the previous one. If the magnitude is 2, then there will be one space between; if 3, then two spaces between, etc. Therefore, a value of zero would mean that the next line should be drawn directly over the previous one. Since this would make no sense at all, the value of zero is used to indicate the end of the P(i) sequence.

Now here comes the neat part: If the polygon was not completely filled and a value of P(i) = 0 is obtained (signifying the end of the line spacing sequence), then the line spacing sequence is set back to the first element in the sequence, P(1), and the procedure continues until the polygon is filled up.

Therefore, one of the simplest sequences for this array is P(0) = 1, P(1) = -2, P(2) = 0. For this example sequence P(0) specifies odd parity, P(1) specifies the application of the odd parity to that line and to increment the line position by 2 (i.e., to skip one space), and P(2) ends the sequence. This command sequence will fill the polygon with a series of lines separated by one space, with each line drawn from an odd X coordinate.

When TYPE = 2 for Pixel Painting, additional information must be specified. The variable PB specifies the random probability blend of plotted and unchanged pixels. The plotted pixels are colored as per the previously specified COLOR command. The unchanged pixels may be spoken of as being plotted with the "transparent" color. This gives us the possibility of doing a multiplicity of "overlay" effects on the same area. If PB is less than 1, then PB specifies the probability or proportion of plotted pixels. When PB=.1 then ten percent of the pixels will be plotted. When PB=.5 then 50 percent will be plotted, etc.

If PB=1 then another form of blending will be used, and another variable, *PC*, must be used to control this technique. When PC>0 then the first line drawn will be entirely unchanged pixels and the last line drawn to fill the polygon will be entirely plotted pixels. The lines in between will have a higher proportion of plotted pixels as they are drawn closer to the end of the polygon. When PC<0 then the unchanged-plotted pixel effect is reversed: the first line will be entirely plotted and the last line will be entirely unchanged.

The magnitude of *PC* determines the rate at which the proportion of pixels plotted changes from the start of the polygon to the end. If ABS(PC)=1, then the proportion ratio changes evenly from one end of the polygon to the other. The line drawn at the halfway point will have half of its pixels plotted and the other half unchanged. When ABS(PC)>1, say 2 or 4, the start condition phases out more slowly to the end condition, and the start condition will be seen to have more influence over the entire polygon. When ABS(PC)<1, say 0.5 or 0.25, then the start condition phases out more rapidly and the end condition is seen to dominate the polygon area.

If you have read the previous article, you may have noted that the "checkerboard" option that was available in the previous version is not specified here. This routine can be used to provide that effect if 45 degree lines are drawn every other space over a previously colored polygon.

This concludes a functional description of the POLY8 subroutine. We will take a brief tour of its structure before continuing on to the more practical aspects of how to use it. Overall, a great similarity will be found with respect to the polygon fill subroutines that were previously documented. Since these have been explained in previous sections, we will just focus on the differences here.

The additional and new TYPE options, implemented in lines 18405 to 18425, are fairly obvious. A big functional difference between this routine and the previous one is that this one lets us fill the polygon with lines of any specified slope. In order to implement this feature in the simplest way, a U, V axis system is defined as being rotated with respect to the screen X,Y axis system by the angle whose

Sine and Cosine values are stored in the variables SA and CA, respectively. SA and CA are calculated in terms of the slope ratio RA in lines 18445 and 18450. RA is also the Tangent of the angle. Then, each X(), Y() array element that defines the polygon perimeter is rotated into a corresponding U(), V() array element in lines 18460 to 18475. Using the same method described in the previous article, vertical line segments are calculated to fill the polygon in the U,V plane, in lines 18550 to 18565. These line segments are then rotated back into the X,Y plane in lines 18575 to 18610. The previously described parity logic is implemented in the middle of this, in lines 18580 to 18590.

For the case of "bar painting," the line segments can be simply drawn with the PLOT-DRAWTO command in line 18620. In the case of "pixel painting," where each pixel must be tested to be plotted or passed over, I finally broke down and did some assembly programming. In Mode 8 graphics there are too many pixels and the coding in BASIC is too slow. The pixel-painting probability factor is calculated in lines 18643 and 18645, and the LINEP subroutine that does the job starts at line 18790.

Note that the machine code component of this routine must first be initialized by GOSUBing to line 18900.

Before we leave these dungeons of program structure, I have a confession to make. With all this geometry-math-programming, I couldn't escape the need for a cut-and-try Finagle correction factor. That's the Z5 round-up factor in lines 18575, 18595 and 18600. It is defined in lines 18430 and 18435. These values generally work well. But if you strike out on your own in the uncharted land of Mode 8 patterns and textures, and you find that for some reason the pattern suddenly shifts in the middle of the polygon, you might want to try modifying that factor for your case.

So much for the POLY8 structure. (Do I hear a sigh of relief somewhere?) The easiest way I know to become familiar with this subroutine is to use it to explore Mode 8 graphics, which is what this is all about anyway! And the easiest way I know to do this is by using Program 4, the PALETT8 program.

Studying the listing of the PALETT8 program, lines 270 and 275 define the constants for often used numeric values to reduce program memory requirements. (Naturally, if the REM statements are eliminated, memory requirements will be reduced much further.) The required DIMensioned arrays are defined in line 280, and the machine code routine is initialized in line 285.

Lines 295 to 320 initialize the E, F scale factors for the desired graphics mode. While these programs were developed specifically for Mode 8, they can be used in any mode. This version of the PALETT8 program can immediately run in Modes 6, 7, and 8. You can modify it to run in the other modes by inserting additional logic in this area.

Lines 370 to 490 set the initial hue-luminance values in the four registers of interest.

It might be useful to pause and study lines 570 to 780, because they represent an interesting programming technique. Instead of cluttering up our program with a group of PLOT-DRAWTO commands, the equivalent coding is set into DATA statements. Then, in lines 750 to 780 the information is used to either PLOT or DRAWTO, depending upon whether the code number is -1 or -2. The negative values for the code make it easy to differentiate the code from the X, Y coordinates in the DATA statements.

We will be studying Mode 8 colors and textures by placing them in one of 15 available screen areas. These square areas are defined in lines 810 to 950 in terms of four numbers that represent the lower left and upper right corners of the squares in the variables XL, YL, XU, YU. In line 1050 the program asks us which AREA we wish to use. If we return a value of from 1 to 15, then line 1090 points to the appropriate DATA location. Line 1100 reads the data, and lines 1110 to 1140 dump the appropriate values in the X(), Y() arrays (scaled appropriately for the specified graphics mode) that the POLY8 routine needs to define its polygon perimeter.

Note that if we return a zero value to the AREA question, then we are shunted to line 1660 where we can change our SETCOLOR assignment values by using the joystick. It works the same as the option in the previous article, as per the REM statements in lines 1840 to 1870.

Note also, as per REM lines 980 to 1030, that RETURNing the value 99 to any question will always reset the input sequence back to the AREA request on line 1050. We.can also do the same thing by pressing the BREAK key and RETURNing the command GOTO 1050.

Now suppose we LOAD the program and RUN it. We specify Mode 8 graphics and, in response to the AREA question, we answer 1. We are then asked to choose between the CREATE and PRESET options by RETURNing 1 or 2. Let me defer an explanation of the PRESET option for the time being, and go straight to the CREATE option by RETURNing a 1.

We are then asked which TYPE of painting we desire, Bar, Pixel, or COLOR. Strictly speaking, the COLOR option is not a painting method at all, just the option to redefine the color register selection for the next PLOT-DRAWTO commands. This option is important, and it was a convenient place to stick it in the question sequence. If we do select the COLOR option, we are asked what COLOR number we desire (line 1600) and are reminded that in Mode 8 only 1 and 0 are functional COLOR numbers. However, we are not restricted from RETURNing any value to obtain the flexibility of being able to use this program in the other graphics modes.

Suppose we select Bar Painting. We are then asked for the angle ratio. We will select 0 for vertical lines. We are then asked for the value of the parity variable, P(0). We will select 1 for odd parity. Then we are asked for a value to P(1). Note that the program will refuse the value 0 for this variable. We will return with the value -2, which specifies the use of the parity check because it is negative, and will move two spaces (i.e., skip one space). We are then asked for the value of P(1). We return a value of 0 to end the sequence, and watch while it fills AREA 1 with Mode 8 primary blue (brown).

We are asked again "which AREA?" If we answer with a 2 and then repeat the exact same answers we gave previously (except for the parity variable P(0), for which we will return a value of 2 for even parity), we will see the machine fill area 2 with Mode 8 primary brown (blue).

With the next two areas we can repeat what we have just done. But by returning a value of 2 for the angle ratio we will obtain the Mode 8 primary colors, red and green. By returning a value of 1 for the angle ratio, zero for P(0) (no parity check), 2 for P(1), and 1 for P(2) we obtain Mode 8 primary grey.

Now let's go back to our blue area. In response to the AREA request we return a value of 1. This time, on top of the primary blue that is already there, we will overwrite the Mode 8 primary grey pattern and see how it interacts to form blue-grey diagonal bars. If we repeat this over the primary brown, we will see green-grey diagonal bars.

Remember the beginning of this article, where we changed the color of the blue vertical line by drawing another line immediately adjacent to it? Well, let's repeat that here.

We can return an AREA number 10, specify the CREATE mode, Bar Painting, an angle ratio of 0, and parity P(0) of 1. Then, for P(1) we can specify a -1 to move one space with parity check. For P(2) we can specify 3, to move three spaces without parity check. Since we are moving an odd number of spaces from our last parity check, the parity here must be the opposite and should not be checked against the reference P(0) specification. We can end this sequence by returning a value for P(3) of 0, and then watch green (red) vertical bars being drawn.

As we can see, this activity of generating Mode 8 textures and colors can go on indefinitely. Now let's take a look at that PRESET option in the program. However, if you have only 24K of memory, you may have to delete this PRESET coding, or at least some of the PRESET DATA lines, in order to RUN this program.

From line 1160, in response to returning a value of 2 for the PRESET option, we are shunted to line 1900 and asked "Which PRESET Number?" In this program version, starting at line 2070, we have stored only 67 "PRESET" colors and textures. Line 1930 prohibits a return value of less than 0 or more than 66.

Suppose we return a value of 2 for the PRESET Number. The DATA pointer on line 1950 will then point to DATA line 2090. The READ statement on line 1960 will then pick up the value of 1 on line 2090 and assign it to the TYPE variable. It then proceeds to read three more DATA values in line 1990 and assigns them to the variables RA=0, P(0)=1, and P(1)=-2. It then reads P(2)=0 in line 2010 and, because of its zero value, it ends the READ sequence in line 2020 by GOSUBing to the POLY8 subroutine where it will proceed to fill the AREA specified with Mode 8 primary blue (brown).

When it returns from the POLY8 subroutine, it jumps to line 2040 and reads the next numeric DATA on line 2090. In this case the value is zero so the program loops back to the AREA question. However, if the value was 99 the program would have looped back to read the next value for TYPE, plus the rest of the information that would have laid another pattern over the previous one.

PRESET number 0, DATA line 2070, is important because it can be used as an eraser to wipe out previous patterns. It consists of a TYPE 7 COLOR command, a COLOR value of 0 (blank out), 99 to loop around for another READ, a new TYPE=1 (Bar Painting), RATIO=0 (vertical), P(0)=0 (no parity), P(1)=1 (no spaces between lines), P(2)=0 to end the sequence and plot the off pixels, 99 to loop around again, 7 for the COLOR command, 1 to turn the COLOR back on, and 0 to end the DATA sequence and jump to the AREA request.

At this point you are on your own. You can call up the various 66 PRESET patterns stored here and study them. You can create new patterns and, when you find something you want to file, add it to this PRESET sequence. (Be sure to update the input limit test in line 1930.)

If you want to just continue drawing patterns over patterns, as per the Video Easel game, you can modify this program to automate that kind of operation. However, the COLOR should be alternated between 1 and 0, else very quickly it will just be drawing white on white.

At this point I imagine somebody is muttering that all this is nice, but what can we do with it besides play Video Easel type games?

Ah hah! Glad you asked. That PRESET data we assembled was not just an academic study. This is our color-texture PALETTE from which we shall judiciously extract what we need to create scenes with a quality that is unique to the graphics of Mode 8.

The time has now come to take a look at Program 5, the PICTUR8 program. This program is specifically designed for drawing a picture in Mode 8 graphics, although it would be a simple task to modify it for any other graphics mode. The specific example picture in this version of the program is defined entirely within the PRINT and DATA statements starting at line 1000. The program statements before line 1000 are fairly general, with the exception of the Graphics mode and SETCOLOR assignment values. If another Graphics mode is used, where the COLOR register values could be more than just 0 or 1, then the error detection statements in lines 390 and 590 must be changed.

Since some have the CTIA chip and others have the GTIA chip, this program has been designed to display the proper colors in either case. To do this, the program asks which chip is being used at line 330.

A great simplification technique is used here, where the picture DATA is separated into two major functions: AREA and FILL. The AREA DATA starts at line 3000 and defines the geometry in terms of X, Y coordinates. The FILL DATA starts at line 2000 and determines the type of painting technique, and the color-texture pattern that we will use to fill the polygon AREA.

At line 360 in the program the first numeric value is read from a line of AREA DATA and is assigned to the variable FILL. If FILL = 999 then the picture is complete and the program stops reading data and jumps to line 1000.

If the variable FILL is a positive number, then the program will eventually use it to read the appropriate fill data using the program logic at lines 550 and 560. For most cases this will probably be the condition. However, there are certain cases where this AREA-FILL split-up becomes excessively cumbersome and artificial. Therefore, a provision has been made to allow one to bypass the reading of separate fill data for these cases. This bypass option is enacted when the numeric value read for the FILL variable is negative.

For example, line 380 allows the program to fall through if FILL = 7 and to read the next data value as the variable RA. It then assigns the positive value of FILL to the variable type and lets the POLY8 subroutine handle it directly in line 405.

Another special provision has been implemented, allowing the use of multiple data sequences on the same line. After it returns from the POLY8 routine, the program reads the next data value at line 410. If it reads a value of 99, then that tells it to go back and read another set of data without incrementing the AREA DATA pointer. Otherwise, it will increment the AREA DATA pointer in lines 340 and 350 before reading the next data set.

If the variable FILL is not -7 then it reads the next data value and assigns it to the variable NP, at line 470. The program then proceeds to read in the next NP amount of number pairs which it promptly stores in the X(), Y() arrays, at line 496.

On line 510 FILL is checked one more time and, if it is a negative number, TYPE is set equal to that positive value of FILL and the program GOSUBs directly to the POLY8 subroutine. Otherwise, the program reads the value for the type variable from the fill DATA statement, as per line 550 and 560. If TYPE = 7 then it simply reads the next data value and sets the color in line 600. It then jumps to line 700 where it reads the next data value. Here, too, a value of 99 will enable it to keep reading fill data without jumping back to the AREA DATA and incrementing the AREA pointer.

If TYPE is not equal to 7, then it checks whether TYPE = 2, at line 620. If it is, then the next two data values are assigned to the variables PB and PC, for the POLY8 routine. If TYPE is not equal to 2, then the program directly reads the values for the variables RA, P(0), P(1), P(2) etc., and then GOSUBs to the POLY8 routine.

Suppose we take a look at some of the data and see how the program uses it. The AREA DATA starts at line 3000. Originally there were two items to be painted by the data that was in lines 3010 and 3020. Then I decided to move these items down in the sequence. Instead of changing all the line numbers of the data statements, I just inserted two "dummy" statements that don't really change anything. They both just specify a -7 code for the FILL variable, which tells the program to change the COLOR assignment. The next number is 1, so the program just implements the command COLOR 1, which was previously done anyway. The final zero in the data statement terminates the sequence. The remaining alphanumeric string is stored as if it were string data, but it is never read and has been inserted only for REMark type information.

The first real area painting is done in line 3030. The end string tells us that this is AREA 3 DATA, and that it is supposed to represent city walls. The first number, 8, is the value for the variable FILL. This will cause the program to read the fill data set at line 2080, which is "Bar Red," according to its alphanumeric label. The first number in line 2080 sets TYPE = 1 for Bar Painting. The second number sets RA = 0 for vertical lines. The third number sets P(0) = 2 for even parity check. The next two numbers set P(1) = -1 to move one space with parity check, and P(2) = 3 to then move 3 spaces without parity check. The next number sets P(3) = 0 to end the move command sequence. The final number of zero ends the DATA sequence for that line.

Going back to line 3030, the second number sets NP = 4, and the next eight numbers set the four pair of X, Y values into the X(), Y() arrays. The final zero ends the DATA sequence for that line.

The next area data on line 3040 has a similar format, but it draws us a blue ocean with the fill data at line 2020. The area data at line 3050 is interesting. We can see that it has exactly the same numbers as the ocean area data, except for its first FILL number of 9. It is covering exactly the same screen area as the "ocean" data, but this data will provide a watery looking reflection effect. FILL DATA number 9 is called "receding bars." It has TYPE = 2 for Pixel Painting. Thus, the next two numbers must define PB = 1 and PC = -1. The value of PB = 1 specifies that there will be an uneven pixel-on probability across the polygon, and that PC must be used to specify it. The value of PC = -1 specifies that the initial line will be drawn with all pixels on, the final line will be drawn with all pixels off, and that the pixel proportion will vary evenly in between. An RA = 100 specifies horizontal lines, a P(0) = 0 specifies no parity check, and the remaining lines specify the number of spaces.

This example is interesting for a number of reasons. First, unlike the other cases, the line spacing sequence is non-repetitive. Second, the optical effect of distance is achieved in two ways: the changing line spacing, and the changing pixel-on proportions.

The next three area data statements are interesting because they all work the same screen area, but with a different FILL specification. First, line 3060 calls for FILL = 0, which is the fill data at line 2000. This data is called the "horizontal eraser." The first value at line 2000 specifies TYPE = 7, RA = 0, which turns COLOR off. Then a 99 continue leads to TYPE = 1 Bar Painting, RA = 100 horizontal lines, P(0) = 0, P(1) = 1, P(2) = 0 for no parity check, lines drawn immediately adjacent with no spaces in between. Another 99 to continue, and then TYPE = 7, RA = 1 to reset the COLOR register for further painting.

Unlike the lower resolution modes where we can simply over-plot previously-colored pixels with new pixels colored differently, in Mode 8 graphics all the previously colored pixels must be turned off if we want to insert a specific color pattern.

Line 3070 then inserts the primary color-texture pattern for this area, and line 3080 then adds, on top of it, some additional random texturizing effect to eliminate a flat repetitiveness.

The full power of Mode 8 hi-resolution graphics is called for in lines 3180 to 3220. Lines 3180 and 3190 draw the small sailboat, where the Mode 8 pixel size is needed to provide the proper proportions for this small image.

Then line 3200 turns the color register off with a -7,0 command. The 99 continue then leads to a value of -4, which specifies a direct TYPE = 4 command to just PLOT individual X, Y pixel coordinates. An NP = 10 specifies the 10 X, Y coordinate pairs to be PLOTted in the boat, to represent the two individuals. Then on line 3210 the COLOR register is turned back on.

The boat wake is drawn with line 3220 data which again turns the COLOR off with the -7,0 data values. The 99 continue leads to a -5 for a direct TYPE = 5 command to just PLOT-DRAWTO lines, and the NP = 8 value specifies the eight X, Y pairs to draw the four lines.

Of course, the mountain itself is the biggest surprise. But I won't spoil it for you by describing it here. Run it and see for yourself. With the previously analyzed sections as background, you should be able to decipher the DATA code for these last elements with little difficulty.

There are quite a few game programs for sale that advertise the use of high resolution graphics with much enthusiasm. These include adventure programs, space programs, and utility programs. Often these are advertised in the magazines, with multi-colored dramatic pictures. However, when we actually see how they use Mode 8 to illustrate their programs, it usually doesn't even come close to the dramatic pictures in the advertisements. If you have run this program and seen this sample picture, then you know the programs presented here can be used to close that gap between the drama of the advertising illustrations and the drama of the program illustrations.

It may be of interest to note that Paul Twitchell's book, *The Tiger's Fang,* was a most appropriate source of inspiration to demonstrate these dramatic graphic techniques. This book is the story of one of the greatest adventures of all time.

One very useful TYPE option which was not demonstrated in this example picture is the case where TYPE = 3, to just draw the boundary of an area. This option can be used in the initial picture development phase to outline the area boundaries.

Naturally, we will start creating our overall picture on a grid paper, with the screen coordinates marked off on the horizontal and vertical edges. We sketch out our picture and then outline our major areas with a straightedge and note the X, Y vertex coordinates of the perimeters. High-resolution details must be "blown up" on another sheet of grid paper so we can see each individual pixel of their form.

Once we have a good layout for our picture geometry, then we can start constructing our AREA DATA statements after line 3000, in line increments of 10. If we specify the first numeric value to be -3, then we will see what our geometric outlines look like on the screen without using any fill DATA statements.

You can see what this looks like by changing the first numeric value to -3 in all the DATA statements from line 3030 to 3190 in the PICTUR8 program.

You are now on your own to use these programs or modify them as you see fit.

## PROGRAM 1. Multi-Colored Graphics In Mode 8.

```
18000 REM ============================
18005 REM ={4 SPACES}POLY8 Subroutine
      {5 SPACES}=
18010 REM =  Polygon Painting For{3 SPACES}=
18015 REM = BASIC  Graphics Mode 8   =
18020 REM ============================
18023 REM ={10 SPACES}by;{12 SPACES}=
18024 REM ={7 SPACES}Phil Dunn{9 SPACES}=
18025 REM ={5 SPACES}12 Monroe Ave.
      {6 SPACES}=
18026 REM =  Hicksville, NY 11801{3 SPACES}=
18030 REM ============================
18035 REM Enter with the value for
18040 REM TYPE= Type of Painting
18045 REM "{3 SPACES}= 1 for Bar Painting
18050 REM "{3 SPACES}= 2 for Pixel Painting
18053 REM "{3 SPACES}= 3 for a Line Boundary
18055 REM "{3 SPACES}= 4 to PLOT X(i),Y(i)
18057 REM "{3 SPACES}= 5 TO PLOT X(i),Y(i),
18058 REM .  and DRAWTO X(i+1),Y(i+1)
18060 REM "{3 SPACES}= 6 to DRAWTO X(i),Y(i)
18065 REM "{3 SPACES}= 7 to COLOR RA
18070 REM ............................
18075 REM For TYPE = 1 to 6,
18080 REM enter with the values...
18085 REM NP = No. of Vertex Points.
18087 REM .  for i=1 to NP:
18090 REM X(i)= DIM Vertex X values
18095 REM Y(i)= DIM Vertex Y values
18100 REM ............................
18105 REM For TYPE 1 and 2 Painting
18110 REM also enter values for...
18115 REM RA = Angle X/Y Ratio
18120 REM "   = 0 for Vertical
18125 REM "   = +-1 for +-45 degrees
18130 REM "   = 100 for Horizontal
18135 REM P()= DIM array for spacing
18140 REM .Parity Color-Lock Option:
18145 REM .P(0)= 0 for no parity
18150 REM .P(0)= 1 for odd parity
18155 REM .P(0)= 2 for even parity
18160 REM .for i=1 to something:
18165 REM .ABS(P(i))= Spaces to move
18170 REM .SGN(P(i))=+1, no parity
18175 REM .SGN(P(i))=-1, parity lock
18180 REM .{4 SPACES}P(i) = 0 to end data
18185 REM ............................
18190 REM For TYPE 2 Pixel Painting
```

```
18195 REM also enter values for...
18200 REM PB & PC For Pixel Blending
18205 REM * Set PB<1 for an even
18210 REM . blend of active and
18215 REM . inactive pixels.
18220 REM . PB= the proportion of
18225 REM . active pixels.  PB>0
18230 REM * Set PB>=1 for an uneven
18235 REM . blend across the area.
18240 REM . area.  Then...
18245 REM .  If PC>0 then the start
18250 REM .{4 SPACES}color is inactive and
18255 REM .{4 SPACES}the end is active.
18260 REM .  If PC<0 then the start
18265 REM .{4 SPACES}color is active and
18270 REM .{4 SPACES}the end is inactive.
18275 REM .  When ABS(PC)=1 then the
18280 REM .{4 SPACES}start color phases out
18285 REM .{4 SPACES}evenly to the end.
18290 REM .  When ABS(PC)>1 then the
18295 REM .{4 SPACES}start color phases out
18300 REM .{4 SPACES}more slowly.
18305 REM .  When ABS(PC)<1 then the
18310 REM .{4 SPACES}start color phases out
18315 REM .{4 SPACES}more rapidly.
18320 REM ...........................
18325 REM For TYPE = 7, to COLOR RA,
18330 REM enter with RA = 0 or 1
18335 REM ...........................
18340 REM S()= DIM array used here
18345 REM T()= DIM array used here
18350 REM U()= DIM array used here
18355 REM V()= DIM array used here
18360 REM Variable names used...
18365 REM C0,C1,C2,Z3,Z4,Z5,Z9,
18370 REM X1,Y1,X2,Y2,CA,SA
18375 REM K,L,M,N,IM,IP,MAX,MIN,NOW
18380 REM ===========================
18400 C0=0:C1=1:C2=2:Z9=999
18405 IF TYPE=3 THEN PLOT X(C1),Y(C1):FOR N=
      C2 TO NP:DRAWTO X(N),Y(N):NEXT N:DRAWT
      O X(C1),Y(C1):RETURN
18410 IF TYPE=4 THEN FOR N=C1 TO NP:PLOT X(N
      ),Y(N):NEXT N:RETURN
18415 IF TYPE=5 THEN FOR N=C1 TO NP STEP C2:
      PLOT X(N),Y(N):DRAWTO X(N+C1),Y(N+C1):
      NEXT N:RETURN
18420 IF TYPE=6 THEN FOR N=C1 TO NP:DRAWTO X
      (N),Y(N):NEXT N:RETURN
18425 IF TYPE=7 THEN COLOR RA:RETURN
```

```
18430 Z5=0.5:IF RA=C2 THEN Z5=0.55
18435 IF RA=-C2 THEN Z5=0.18
18440 Z3=-C1:IF RA<>CO THEN Z3=SGN(RA)
18445 SA=SQR(1/(1+RA^C2))*Z3:IF ABS(SA)<0.2
      THEN SA=CO
18450 CA=SQR(1-SA^C2)
18455 REM Rotate X(),Y() to U(),V():
18460 FOR M=C1 TO NP
18465 U(M)=X(M)*CA+Y(M)*SA
18470 V(M)=-X(M)*SA+Y(M)*CA:NEXT M
18480 FOR M=C1 TO NP:N=M+C1:IF N>NP THEN N=C
      1
18485 REM Calculate slopes S() amd Y axis in
      tercepts T()
18490 IF U(M)=U(N) THEN S(M)=Z9:GOTO 18510
18495 S(M)=(V(N)-V(M))/(U(N)-U(M)):T(M)=V(M)
      -S(M)*U(M)
18500 IF ABS(S(M))>Z9 THEN S(M)=Z9
18505 IF ABS(S(M))<C1/Z9 THEN S(M)=CO
18510 NEXT M:MAX=-Z9:MIN=Z9:FOR M=C1 TO NP:Z
      3=V(M)
18515 IF MAX<Z3 THEN MAX=Z3
18520 IF MIN>Z3 THEN MIN=Z3:N=M
18525 NEXT M:MXMN=MAX-MIN:NOW=MIN:IM=N-C1:IF
       IM<C1 THEN IM=NP
18527 IP=N+C1:IF IP>NP THEN IP=C1
18530 IF P(C1)=CO THEN P(C1)=C1:P(C2)=CO
18535 M=C1:IF P(M)<CO THEN M=C1
18540 GOTO 18675
18545 REM Calculate intercepts...
18550 IF S(N)=Z9 OR S(N)=CO THEN Z3=U(N):GOT
      O 18560
18555 Z3=(NOW-T(N))/S(N)
18560 IF S(IM)=Z9 OR S(IM)=CO THEN Z4=U(IM):
      GOTO 18575
18565 Z4=(NOW-T(IM))/S(IM)
18570 REM Rotate U(),V() to X(),Y():
18575 X1=INT(Z3*CA-NOW*SA+Z5)
18580 IF NOW<>MIN AND P(M)>=CO THEN 18595
18585 IF P(CO)=C1 THEN IF X1=C2*INT(X1/C2) T
      HEN X1=X1+C1
18590 IF P(CO)=C2 THEN IF X1<>C2*INT(X1/C2)
      THEN X1=X1+C1
18595 Y1=INT(Z3*SA+NOW*CA+Z5)
18600 Y2=INT(Z4*SA+NOW*CA+Z5)
18605 IF SA=CO THEN X2=Z4*CA
18610 IF SA<>CO THEN X2=X1+(Y2-Y1)*RA
18615 REM Bar Painting...............
18620 IF TYPE=1 THEN PLOT X1,Y1:DRAWTO X2,Y2
      :GOTO 18660
```

```
18635 REM Pixel Painting............
18643 IF PB<C1 THEN PR=PB:GOTO 18650
18645 PR=((NOW-MIN)/MXMN)^ABS(PC):IF PC<CO T
      HEN PR=C1-PR
18650 GOSUB LINEP
18655 REM ............................
18660 NOW=Y1*CA-X1*SA
18665 REM Increment NOW for next bar
18670 NOW=NOW+ABS(P(M)):M=M+C1:IF P(M)=CO TH
      EN M=C1
18675 IF NOW<V(IP) THEN 18690
18680 IF V(IP)=MAX THEN RETURN
18685 N=IP:IP=IP+C1:IF IP>NP THEN IP=C1
18690 IF NOW<V(IM) THEN GOTO 18550
18695 IF V(IM)=MAX THEN RETURN
18700 IM=IM-C1:IF IM<C1 THEN IM=NP
18705 GOTO 18550
18710 REM ============================
```

## PROGRAM 2. Multi-Colored Graphics In Mode 8.

```
18750 REM LINEP Subroutine
18751 REM by... Phil Dunn
18755 REM Draws a line pixel by pixel
18760 REM With a probability to plot
18765 REM or skip each pixel.
18770 REM X1,Y1 = start point
18775 REM X2,Y2 = end point
18780 REM PR = probability to PLOT
18785 REM "  >=0 and <=1
18790 C0=0:C1=1:C255=255:C230=230:C198=198
18795 C92=92:C97=97:C103=103:C106=106:C120=1
      20
18800 K=X2-X1:L=Y2-Y1:Z3=ABS(K):Z4=ABS(L):BI
      =PR*C255
18805 IF K=C0 AND L=C0 THEN RETURN
18810 IF K<C0 THEN 18840
18815 LINE$(C92,C92)=CHR$(C230):REM INC
18820 LINE$(C97,C97)=CHR$(C0)
18825 LINE$(C103,C103)=CHR$(C1)
18830 LINE$(C106,C106)=CHR$(C230)
18835 GOTO 18860
18840 LINE$(C92,C92)=CHR$(C198):REM DEC
18845 LINE$(C97,C97)=CHR$(C255)
18850 LINE$(C103,C103)=CHR$(C0)
18855 LINE$(C106,C106)=CHR$(C198)
18860 LINE$(C120,C120)=CHR$(C230):IF L<C0 TH
      EN LINE$(C120,C120)=CHR$(C198)
18865 IF Z3=C0 THEN Z3=C1/C255
18870 IF Z4=C0 THEN Z4=C1/C255
18875 IF Z3>=Z4 THEN K=Z3:Y2=Z3/Z4:X2=1:IF Y
      2>C255 THEN Y2=C255
18880 IF Z3<Z4 THEN K=Z4:X2=Z4/Z3:Y2=1:IF X2
      >C255 THEN X2=C255
18883 IF K<1 THEN RETURN
18884 POKE 752,C1
18885 L=USR(ADR(LINE$),K,X1,Y1,X2,Y2,BI)
18890 RETURN
18895 REM ...........................
18900 DIM LINE$(136):RESTORE 18910:LINEP=187
      90
18905 FOR K=1 TO 136:READ L:LINE$(K,K)=CHR$(
      L):NEXT K:RETURN
18910 DATA 104,104,133,211,104,133,210,104,1
      33,86
18915 DATA 104,133,85,104,104,133,84,104,104
      ,133
18920 DATA 209,170,104,104,133,208,168,104,1
      04,133
```

```
18925 DATA 207,173,10,210,24,101,207,144,46,
      138
18930 DATA 72,152,72,165,86,72,165,85,72,165
18935 DATA 84,72,162,96,169,11,157,66,3,169
18940 DATA 0,157,72,3,157,73,3,169,1,32
18945 DATA 86,228,104,133,84,104,133,85,104,
      133
18950 DATA 86,104,168,104,170,202,208,19,165
      ,209
18955 DATA 170,230,85,165,85,201,0,208,8,165
18960 DATA 86,201,1,240,30,230,86,169,255,19
      7
18965 DATA 208,240,8,136,208,5,165,208,168,2
      30
18970 DATA 84,198,210,208,162,169,0,197,211,
      240
18975 DATA 4,198,211,240,152,96
```

## PROGRAM 3. Multi-Colored Graphics In Mode 8.

```
0100  *=$5000
0110 ;A=USR(LOC,N,X,Y,NX,NY,BI)
0120 ;DRAWS A LINE WITH EACH PIXEL
0130 ;PROBABILITY-TESTED TO BE
0140 ;ILLUMINATED OR NOT
0150 ;
0160 NH=$D3       ;NO.POINTS-HI BYTE
0170 NL=$D2       ;NO.POINTS-LO BYTE
0180 XH=$56       ;CURSOR X -HI BYTE
0190 XL=$55       ;CURSOR X -LO BYTE
0200 Y=$54        ;CURSOR Y
0210 NX=$D1       ;X COUNT
0220 NY=$D0       ;Y COUNT
0230 BI=$CF       ;PROB. BIAS
0240 RANDOM=$D20A;RANDOM # = 0-255
0250 CIO=$E456     ;CENTRAL I/O
0260 ICCOM=$342    ;IoCb COMmand to CIO
0270 ICBLEN=$348 ;IoCb Buffer LENgth
0280 CPBINR=$B ;Comnd.Put BINary Rec.
0290 ;
0300  PLA          ;CLEAR STACK
0310  PLA
0320  STA NH       ;N - HIGH BYTE
0330  PLA
0340  STA NL       ;N - LOW BYTE
0350  PLA
0360  STA XH       ;X -HIGH BYTE
0370  PLA
0380  STA XL       ;X -LOW BYTE
0390  PLA
0400  PLA
0410  STA Y        ;Y -LOW BYTE
0420  PLA
0430  PLA
0440  STA NX       ;X COUNTER
0450  TAX          ;TEMP X COUNTER
0460  PLA
0470  PLA
0480  STA NY       ;Y COUNTER
0490  TAY          ;TEMP Y COUNTER
0500  PLA
0510  PLA
0520  STA BI       ;PROB. BIAS
0530 ;
0540 LOOP LDA RANDOM
0550  CLC          ;CLEAR CARRY
0560  ADC BI       ;ADD BIAS W. CARRY
0570  BCC DOX      ;BYP.IF CARRY NOTSET
```

```
0580  TXA
0590  PHA           ;SAVE TEMP X
0600  TYA
0610  PHA           ;SAVE TEMP Y
0620  LDA XH
0630  PHA           ;SAVE CURSOR XH
0640  LDA XL
0650  PHA           ;SAVE CURSOR XL
0660  LDA Y
0670  PHA           ;SAVE CURSOR Y
0680  LDX #6*$10 ;OFFSET TO IOCB#6
0690  LDA #CPBINR;Comd.Put BINary Rec
0700  STA ICCOM,X
0710  LDA #0
0720  STA ICBLEN,X
0730  STA ICBLEN+1,X
0740  LDA #1        ;COLOR Reg.#
0750  JSR CIO       ;Use CIO to PLOT
0760  PLA
0770  STA Y         ;CLAIM CURSOR Y
0780  PLA
0790  STA XL        ;CLAIM CURSOR XL
0800  PLA
0810  STA XH        ;CLAIM CURSOR XH
0820  PLA
0830  TAY           ;CLAIM TEMP Y
0840  PLA
0850  TAX           ;CLAIM TEMP X
0860 ;
0870 DOX DEX        ;DECR. TEMP. X
0880  BNE DOY       ;BYP.& DO Y IF<>0
0890  LDA NX
0900  TAX           ;RESET TEMP. X
0910  INC XL        ;INCR. CURSOR X LOC
0920  LDA XL        ;LOAD ACCUM. W. XL
0930  CMP #$00      ;COMPARE WITH 0
0940  BNE DOY       ;BYP. IF NOT SAME
0950  LDA XH
0960  CMP #$01      ;IS HI BYTE=1?
0970  BEQ RTS       ;IF IT IS, RETURN
0980  INC XH        ;INC. HI BYTE
0990 ;
1000 DOY LDA #$FF
1010  CMP NY        ;COMPARE NY WITH 255
1020  BEQ COUN      ;BYP.IF=0 (HOR.LINE)
1030  DEY           ;DEC.TEMP Y
1040  BNE COUN      ;BYP.& DO COUN IF<>0
1050  LDA NY
1060  TAY           ;RESET TEMP. Y
1070  INC Y         ;SHIFT CURSOR Y LOC
```

```
1080 ;
1090 COUN DEC NL ;COUNT PIXEL NO.
1100  BNE LOOP    ;KEEP PLOTTING IF<>0
1110  LDA #$00
1120  CMP NH      ;IS NH ZERO?
1130  BEQ RTS     ;IF IT IS, RETURN
1140  DEC NH      ;DECREMENT NH
1150  BEQ LOOP    ;        & LOOP
1160 RTS RTS      ;RETURN
1170  .END
```

## PROGRAM 4. Multi-Colored Graphics In Mode 8.

```
100 GRAPHICS 0
110 ? " ==============================="
120 ? " ={9 SPACES}PALETT8{11 SPACES}="
130 ? " ={5 SPACES}A Color-Texture
    {7 SPACES}="
140 ? " ={3 SPACES}Development  Program
    {4 SPACES}="
150 ? " = For The POLY8 Subroutine  ="
160 ? " ==============================="
170 REM ={11 SPACES}by:{13 SPACES}=
180 REM ={8 SPACES}Phil Dunn{10 SPACES}=
190 REM ={6 SPACES}12 Monroe Ave.{7 SPACES}=
200 REM ={3 SPACES}Hicksville, NY 11801
    {4 SPACES}=
210 REM ==============================
270 C0=0:C1=1:C2=2:C3=3:C4=4:C5=5:C6=6:C7=7:
    C8=8:C9=9:C10=10
275 C99=99:AREA=1050:NP=C4:POLY8=18400:DETRA
    P=40000
280 DIM X(C4),Y(C4),P(C10),S(C4),T(C4),U(C4)
    ,V(C4),CO(C3),LU(C3)
284 ? :? :? :? "{4 SPACES}Reading data..."
285 GOSUB 18900:REM SETUP LINEP SUB.
290 REM ==============================
295 ? :? :? "GRAPHICS MODE (6 to 8) = ";
300 TRAP 295:INPUT MODE:TRAP DETRAP
305 IF MODE<C6 OR MODE>C8 THEN 295
310 GRAPHICS MODE
315 IF MODE=C6 OR MODE=C7 THEN E=0.49:F=0.49
320 IF MODE=C8 THEN E=C1:F=C1
330 REM ==============================
360 REM Initial color assignment...
370 CO(C0)=C0:LU(C0)=C0
380 CO(C1)=C2:LU(C1)=C8
390 CO(C2)=12:LU(C2)=C10
400 CO(C3)=C9:LU(C3)=C4
410 IF MODE<>C8 THEN 500
420 CO(C0)=C0:LU(C0)=C0
430 CO(C1)=C0:LU(C1)=C0
440 CO(C2)=C0:LU(C2)=12
450 CO(C3)=C0:LU(C3)=C0
460 SETCOLOR C0,CO(C1),LU(C1)
470 SETCOLOR C1,CO(C2),LU(C2):REM GRAPHC
480 SETCOLOR C2,CO(C3),LU(C3):REM BACKGR
490 SETCOLOR C4,CO(C0),LU(C0):REM BORDER
500 COLOR C1
510 REM ...........................
520 REM DATA for area numbers:
```

```
530 REM -1 = PLOT
540 REM -2 = DRAWTO
550 REM -9 = end of data
560 REM One number per DATA statement
570 DATA -1,5,20,-2,5,40
580 DATA -1,65,20,-2,70,20,-2,65,40,-2,70,40
590 DATA -1,130,20,-2,135,20,-2,135,40,-2,13
    0,40,-1,130,30,-2,135,30
600 DATA -1,200,40,-2,200,20,-2,195,30,-2,20
    2,30
610 DATA -1,265,20,-2,260,20,-2,260,30,-2,26
    5,30,-2,265,40,-2,260,40
620 DATA -1,2,70,-2,2,90,-2,7,90,-2,7,80,-2,
    2,80
630 DATA -1,65,70,-2,70,70,-2,65,90
640 DATA -1,130,70,-2,135,70,-2,130,90,-2,13
    5,90,-2,130,70
650 DATA -1,200,90,-2,200,70,-2,195,70,-2,19
    5,80,-2,200,80
660 DATA -1,259,70,-2,259,90,-1,263,70,-2,26
    3,90,-2,267,90,-2,267,70,-2,263,70
670 DATA -1,3,120,-2,3,140,-1,7,120,-2,7,140
680 DATA -1,64,120,-2,64,140,-1,67,120,-2,72
    ,120,-2,67,140,-2,72,140
690 DATA -1,129,120,-2,129,140,-1,132,120,-2
    ,137,120,-2,137,140,-2,132,140,-1,132,13
    0,-2,137,130
700 DATA -1,194,120,-2,194,140,-1,200,140,-2
    ,200,120,-2,197,130,-2,202,130
710 DATA -1,259,120,-2,259,140,-1,267,120,-2
    ,263,120,-2,263,130,-2,267,130,-2,267,14
    0,-2,263,140
720 DATA -9,-9,-9
730 REM ..............................
740 REM Read DATA and draw numbers
745 RESTORE 570
750 READ I,X,Y
760 IF I=-C1 THEN PLOT X*E,Y*F
770 IF I=-C2 THEN DRAWTO X*E,Y*F
780 IF I<>-C9 THEN 750
790 REM ..............................
800 REM DATA for areas: XL,YL,XU,YU
810 DATA 10,45,60,2
820 DATA 75,45,125,2
830 DATA 140,45,190,2
840 DATA 205,45,255,2
850 DATA 270,45,317,2
860 DATA 10,95,60,50
870 DATA 75,95,125,50
880 DATA 140,95,190,50
```

```
890 DATA 205,95,255,50
900 DATA 270,95,317,50
910 DATA 10,145,60,100
920 DATA 75,145,125,100
930 DATA 140,145,190,100
940 DATA 205,145,255,100
950 DATA 270,145,317,100
960 REM ..............................
970 REM ========================
980 REM = NOTE:  To RESET the  =
990 REM ={3 SPACES}input sequence{5 SPACES}=
1000 REM =  back to the "AREA"  =
1010 REM ={6 SPACES}request{9 SPACES}=
1020 REM = RETURN the value 99  =
1030 REM ={3 SPACES}to any question
     {4 SPACES}=
1040 REM ========================
1050 ? :? :? "Which AREA (0-15)";
1060 TRAP AREA:INPUT A:TRAP DETRAP
1070 IF A<C0 OR A>15 THEN GOTO AREA
1080 IF A=C0 THEN 1660
1090 RESTORE 800+A*C10
1100 READ XL,YL,XU,YU
1110 X(C1)=XL*E:Y(C1)=YL*F
1120 X(C2)=XL*E:Y(C2)=YU*F
1130 X(C3)=XU*E:Y(C3)=YU*F
1140 X(C4)=XU*E:Y(C4)=YL*F
1150 REM ..............................
1160 ? "CREATE, or PRESET (1 or 2)";
1170 TRAP 1160:INPUT I:TRAP DETRAP
1180 IF I=C99 THEN GOTO AREA
1190 IF I=C2 THEN 1900
1200 IF I<>C1 THEN 1160
1210 REM ..............................
1220 ? "Which TYPE of Painting;"
1230 ? "1=Bar, 2=Pixel, 7=COLOR ";
1240 TRAP 1220:INPUT TYPE:TRAP DETRAP
1250 IF TYPE=C99 THEN GOTO AREA
1260 IF TYPE=C7 THEN 1600
1270 IF TYPE=C1 THEN 1410
1280 IF TYPE<>C2 THEN 1220
1290 REM ===========================
1300 ? "For Pixel Painting,"
1310 ? "What value for PB, (0-1)";
1320 TRAP 1310:INPUT PB:TRAP DETRAP
1330 IF PB=C99 THEN GOTO AREA
1340 IF PB<C0 OR PB>C1 THEN 1310
1350 IF PB<C1 THEN 1420
1360 ? "What value for PC";
1370 TRAP 1360:INPUT PC:TRAP DETRAP
```

```
1380 IF PC=C99 THEN GOTO AREA
1390 GOTO 1420
1400 REM ============================
1410 ? "For Bar Painting,"
1420 ? "What Angle X/Y Ratio";
1430 TRAP 1420:INPUT RA:TRAP DETRAP
1440 IF RA=C99 THEN GOTO AREA
1450 REM ............................
1460 ? "P(0) Parity value (0-2)=";
1470 TRAP 1460:INPUT P:TRAP DETRAP
1480 P(CO)=P:IF P<CO OR P>C2 THEN 1460
1490 IF P=C99 THEN GOTO AREA
1500 REM ............................
1510 I=1
1520 ? "What value for P(";I;") (0=end)";
1530 TRAP 1520:INPUT P:TRAP DETRAP
1540 P(I)=P:IF P=C99 THEN GOTO AREA
1550 IF I=C1 AND P=CO THEN 1520
1560 IF P=CO THEN 1580
1570 I=I+C1:GOTO 1520
1580 GOSUB POLY8:GOTO AREA
1590 REM ============================
1600 ? "What COLOR (MODE 8: 1=on, 0=off)";
1610 TRAP 1600:INPUT RA:TRAP DETRAP
1620 IF RA=C99 THEN GOTO AREA
1630 GOTO 1580
1640 REM ============================
1650 REM REDEFINE THE COLORS...
1660 ? "Reg.No. = Hue, Luminance"
1670 ? "0=";CO(CO);",";LU(CO),"1=";CO(C1);",
     ";LU(C1),"2=";CO(C2);",";LU(C2),"3=";CO
     (C3);",";LU(C3)
1680 ? "Which COLOR Register (0-3) ";
1690 TRAP 1660:INPUT I:TRAP DETRAP
1700 IF I=C99 THEN GOTO AREA
1710 IF I<CO OR I>C3 THEN 1660
1720 K=CO(I):L=LU(I):GOTO 1760
1730 IF STRIG(CO)=CO THEN 1660
1740 IF CO(I)=INT(K) AND LU(I)=C2*INT(L/C2)
     THEN 1800
1750 CO(I)=INT(K):LU(I)=C2*INT(L/C2)
1760 ? "Reg. ";I;" = Color ";CO(I);" , Lumin
     ance ";LU(I)
1770 J=I-C1:IF J<CO THEN J=C4
1780 SETCOLOR J,CO(I),LU(I)
1790 CO(I)=INT(K):LU(I)=C2*INT(L/C2)
1800 IF STICK(CO)=C7 THEN K=K+C1/C10:IF K>16
      THEN K=CO
1810 IF STICK(CO)=11 THEN K=K-C1/C10:IF K<CO
      THEN K=15
```

```
1820 IF STICK(C0)=14 THEN L=L+C2/C10:IF L>16
      THEN L=C0
1830 IF STICK(C0)=13 THEN L=L-C2/C10:IF L<C0
      THEN L=14
1840 REM Joystick 0 controls:
1850 REM Left-Rignt changes hue
1860 REM Foward-Back changes luminance
1870 REM Press Trigger to fix selection
1880 GOTO 1730
1890 REM ============================
1900 ? "Which PRESET Number";
1910 TRAP 1900:INPUT I:TRAP DETRAP
1920 IF I=C99 THEN GOTO AREA
1930 IF I<C0 OR I>66 THEN 1900
1950 RESTORE 2070+I*C10
1960 READ TYPE
1970 IF TYPE=C7 THEN READ P:COLOR P:GOTO 2040
1980 IF TYPE=C2 THEN READ PB,PC
1990 READ RA,P0,P
2000 P(C0)=P0:P(C1)=P:I=C2
2010 READ P:P(I)=P
2020 IF P=C0 THEN GOSUB POLY8:GOTO 2040
2030 I=I+C1:GOTO 2010
2040 READ P:IF P=C99 THEN 1960
2050 GOTO AREA
2060 REM PRESET DATA.................
2065 REM Color lables are accurate
2066 REM only for the CTIA chip, and
2067 REM are not correct for the GTIA
2070 DATA 7,0,99,1,0,0,1,0,99,7,1,0, ERASER=
      0
2080 DATA 1,0,0,1,0,0,WHT=1
2090 DATA 1,0,1,-2,0,0,BLU=2
2100 DATA 1,0,2,-2,0,0,BRN=3
2110 DATA 1,0,0,3,0,0,DARK BRN VRT=4
2120 DATA 1,0,0,1,2,0,0,GRN&ORG VRT=5
2130 DATA 1,0,1,-1,3,0,0,GRN VRT=6
2140 DATA 1,0,2,-1,3,0,0,RED VRT=7
2150 DATA 1,1,0,2,0,0,DK GRY=8
2160 DATA 1,1,0,3,0,0,DK GRY DIA=9
2170 DATA 1,2,1,-2,0,0,GRN=10
2180 DATA 1,2,2,-2,0,0,RED=11
2190 DATA 1,2,1,-1,-2,0,0,LT GRN DIA=12
2200 DATA 1,2,2,-1,-2,0,0,PNK DIA=13
2210 DATA 1,2,1,-2,-3,0,0,GRN DIA=14
2220 DATA 1,2,2,-2,-3,0,0,RED DIA=15
2230 DATA 1,3,0,2,0,0,NBY GRY=16
2240 DATA 1,3,0,1,2,0,0,GRY DIA=17
2250 DATA 1,4,1,-2,0,0,NBY RED=18
2260 DATA 1,4,2,-2,0,0,NBY BLU=19
```

```
2270 DATA 1,4,1,-1,-2,0,0,PNK DIA=20
2280 DATA 1,4,2,-1,-2,0,0,GRN DIA=21
2290 DATA 1,0,1,-2,0,99,1,1,0,2,0,0,PNK-GRY
     DIA=22
2300 DATA 1,0,2,-2,0,99,1,1,0,2,0,0,GRN-GRY
     DIA=23
2310 DATA 1,0,1,-2,0,99,1,1,0,3,0,0,NBY BLU-
     GRY=24
2320 DATA 1,0,2,-2,0,99,1,1,0,3,0,0,NBY GRN-
     GRY=25
2330 DATA 1,0,2,-2,0,99,1,2,1,-2,0,0,PNK-GRY
      =26
2340 DATA 1,0,1,-2,0,99,1,2,2,-2,0,0,BLU-GRY
      =27
2350 DATA 1,0,1,-2,0,99,1,2,2,-1,-2,0,0,PNK-
     GRY DIA=28
2360 DATA 1,0,2,-2,0,99,1,2,2,-1,-2,0,0,BLU-
     GRY DIA=29
2370 DATA 1,0,1,-2,0,99,1,3,1,2,0,0,BLU-GRY
     COR=30
2380 DATA 1,0,2,-2,0,99,1,3,1,2,0,0,GRN-GRY
     COR=31
2390 DATA 1,0,1,-2,0,99,1,4,2,-2,0,0,NUB PNK
     =32
2400 DATA 1,0,2,-2,0,99,1,4,1,-2,0,0,NUB GRN
     =33
2410 DATA 1,1,0,2,0,99,1,-1,0,2,0,0,GRN PLAI
     D=34
2420 DATA 1,1,0,2,0,99,1,1,0,3,0,0,BRN DIA=3
     5
2430 DATA 1,1,0,2,0,99,1,-1,0,3,0,0,RED-GRN-
     BLU DIA=36
2440 DATA 1,1,1,3,0,99,1,-1,1,3,0,0,BLU CHKS
     =37
2450 DATA 1,1,2,3,0,99,1,-1,1,3,0,0,RED CHKS
     =38
2460 DATA 1,1,0,2,0,99,1,-2,1,-2,0,0,GR/RED
     DIA=39
2470 DATA 1,1,0,2,0,99,1,-2,2,-2,0,0,RED/GRN
      DIA=40
2480 DATA 1,1,0,3,0,99,1,-2,1,-2,0,0,GRN GRI
     D=41
2490 DATA 1,1,0,3,0,99,1,-2,2,-2,0,0,RED GRI
     D=42
2500 DATA 1,1,0,2,0,99,1,-2,1,-1,-2,0,0,GRY
     DIA=43
2510 DATA 1,1,0,3,0,99,1,-2,1,-1,-2,0,0,GRN
     KNIT=44
2520 DATA 1,1,0,3,0,99,1,-2,2,-1,-2,0,0,PNK
     KNIT=45
```

```
2530 DATA 1,1,0,2,0,99,1,-3,0,2,0,0,BLU-RED
     HOR=46
2540 DATA 1,1,0,2,0,99,1,3,0,1,2,0,0,BLU-RED
      VRT=47
2550 DATA 1,1,0,3,0,99,1,3,0,2,0,0,PNK-BLU D
     IAM=48
2560 DATA 1,1,0,3,0,99,1,3,0,1,2,0,0,PNK-BLU
      DIAG=49
2570 DATA 1,1,0,2,0,99,1,4,1,-2,0,0,=NBY RED
      DIAG=50
2580 DATA 1,1,0,2,0,99,1,4,2,-2,0,0,=NBY GRN
      DIAG=51
2590 DATA 1,2,1,-2,0,99,1,-2,2,-2,0,0,GRY=52
2600 DATA 1,2,1,-2,0,99,1,3,0,2,0,0,GRN-GRY
     VRT=53
2610 DATA 1,2,2,-2,0,99,1,3,0,2,0,0,PNK-GRY
     VRT=54
2620 DATA 1,2,1,-2,0,99,1,4,1,-2,0,0,NBY GRN
      VRT=55
2630 DATA 1,2,2,-2,0,99,1,4,2,-2,0,0,NBY PNK
      VRT=56
2640 DATA 1,2,1,-2,0,99,1,4,2,-2,0,0,GRN VRT
     =57
2650 DATA 1,2,2,-2,0,99,1,4,1,-2,0,0,PNK VRT
     =58
2660 DATA 1,3,0,2,0,99,1,4,1,-2,0,0,PNK VRT
     FAT=59
2670 DATA 1,3,0,2,0,99,1,4,2,-2,0,0,GRN VRT
     FAT=60
2680 DATA 1,1,1,3,0,99,1,-1,1,3,0,0,DULL BLU
     =61
2690 DATA 1,1,2,3,0,99,1,-1,2,3,0,0,DULL BRN
     =62
2700 DATA 1,3,0,2,3,4,5,7,10,15,25,0,0,RECEE
     DING=63
2710 DATA 2,0.5,1,100,0,2,3,4,5,7,10,15,25,0
     ,0,RECEEDING COLORS=64
2720 DATA 2,0.3,1,0,0,1,0,0,EVEN COLOR BLEND
     =65

2730 DATA 2,1,1,1,0,1,0,0,UNEVEN BLEND=66
18000 REM ===========================
18005 REM ={4 SPACES}POLY8 Subroutine
      {5 SPACES}=
18010 REM =  Polygon Painting For{3 SPACES}=
18015 REM = BASIC  Graphics Mode 8  =
18020 REM ===========================
18023 REM ={10 SPACES}by;{12 SPACES}=
18024 REM ={7 SPACES}Phil Dunn{9 SPACES}=
18025 REM ={5 SPACES}12 Monroe Ave.
      {6 SPACES}=
18026 REM =  Hicksville, NY 11801{3 SPACES}=
18030 REM ===========================
18035 REM Enter with the value for
18040 REM TYPE= Type of Painting
18045 REM "{3 SPACES}= 1 for Bar Painting
18050 REM "{3 SPACES}= 2 for Pixel Painting
18053 REM "{3 SPACES}= 3 for a Line Boundary
18055 REM "{3 SPACES}= 4 to PLOT X(i),Y(i)
18057 REM "{3 SPACES}= 5 TO PLOT X(i),Y(i),
18058 REM .  and DRAWTO X(i+1),Y(i+1)
18060 REM "{3 SPACES}= 6 to DRAWTO X(i),Y(i)
18065 REM "{3 SPACES}= 7 to COLOR RA
18070 REM ...........................
18075 REM For TYPE = 1 to 6,
18080 REM enter with the values...
18085 REM NP = No. of Vertex Points.
18087 REM .  for i=1 to NP:
18090 REM X(i)= DIM Vertex X values
18095 REM Y(i)= DIM Vertex Y values
18100 REM ...........................
18105 REM For TYPE 1 and 2 Painting
18110 REM also enter values for...
18115 REM RA = Angle X/Y Ratio
18120 REM "  = 0 for Vertical
18125 REM "  = +-1 for +-45 degrees
18130 REM "  = 100 for Horizontal
18135 REM P()= DIM array for spacing
18140 REM .Parity Color-Lock Option:
18145 REM .P(0)= 0 for no parity
18150 REM .P(0)= 1 for odd parity
18155 REM .P(0)= 2 for even parity
18160 REM .for i=1 to something:
18165 REM .ABS(P(i))= Spaces to move
18170 REM .SGN(P(i))=+1, no parity
18175 REM .SGN(P(i))=-1, parity lock
18180 REM .{4 SPACES}P(i) = 0 to end data
18185 REM ...........................
18190 REM For TYPE 2 Pixel Painting
18195 REM also enter values for...
18200 REM PB & PC For Pixel Blending
18205 REM * Set PB<1 for an even
18210 REM . blend of active and
18215 REM . inactive pixels.
18220 REM . PB= the proportion of
18225 REM . active pixels.  PB>0
18230 REM * Set PB>=1 for an uneven
18235 REM . blend across the area.
18240 REM . area.  Then...
18245 REM . If PC>0 then the start
```

```
18250 REM .{4 SPACES}color is inactive and
18255 REM .{4 SPACES}the end is active.
18260 REM . If PC<0 then the start
18265 REM .{4 SPACES}color is active and
18270 REM .{4 SPACES}the end is inactive.
18275 REM . When ABS(PC)=1 then the
18280 REM .{4 SPACES}start color phases out
18285 REM .{4 SPACES}evenly to the end.
18290 REM . When ABS(PC)>1 then the
18295 REM .{4 SPACES}start color phases out
18300 REM .{4 SPACES}more slowly.
18305 REM . When ABS(PC)<1 then the
18310 REM .{4 SPACES}start color phases out
18315 REM .{4 SPACES}more rapidly.
18320 REM ..........................
18325 REM For TYPE = 7, to COLOR RA,
18330 REM enter with RA = 0 or 1
18335 REM ..........................
18340 REM S()= DIM array used here
18345 REM T()= DIM array used here
18350 REM U()= DIM array used here
18355 REM V()= DIM array used here
18360 REM Variable names used...
18365 REM CO,C1,C2,Z3,Z4,Z5,Z9,
18370 REM X1,Y1,X2,Y2,CA,SA
18375 REM K,L,M,N,IM,IP,MAX,MIN,NOW
18380 REM ==========================
18400 CO=0:C1=1:C2=2:Z9=999
18405 IF TYPE=3 THEN PLOT X(C1),Y(C1):FOR N=
      C2 TO NP:DRAWTO X(N),Y(N):NEXT N:DRAWT
      O X(C1),Y(C1):RETURN
18410 IF TYPE=4 THEN FOR N=C1 TO NP:PLOT X(N
      ),Y(N):NEXT N:RETURN
18415 IF TYPE=5 THEN FOR N=C1 TO NP STEP C2:
      PLOT X(N),Y(N):DRAWTO X(N+C1),Y(N+C1):
      NEXT N:RETURN
18420 IF TYPE=6 THEN FOR N=C1 TO NP:DRAWTO X
      (N),Y(N):NEXT N:RETURN
18425 IF TYPE=7 THEN COLOR RA:RETURN
18430 Z5=0.5:IF RA=C2 THEN Z5=0.55
18435 IF RA=-C2 THEN Z5=0.18
18440 Z3=-C1:IF RA<>CO THEN Z3=SGN(RA)
18445 SA=SQR(1/(1+RA^C2))*Z3:IF ABS(SA)<0.2
      THEN SA=CO
18450 CA=SQR(1-SA^C2)
18455 REM Rotate X(),Y() to U(),V():
18460 FOR M=C1 TO NP
18465 U(M)=X(M)*CA+Y(M)*SA
18470 V(M)=-X(M)*SA+Y(M)*CA:NEXT M
18480 FOR M=C1 TO NP:N=M+C1:IF N>NP THEN N=C1
```

```
18485 REM Calculate slopes S() amd Y axis in
      tercepts T()
18490 IF U(M)=U(N) THEN S(M)=Z9:GOTO 18510
18495 S(M)=(V(N)-V(M))/(U(N)-U(M)):T(M)=V(M)
      -S(M)*U(M)
18500 IF ABS(S(M))>Z9 THEN S(M)=Z9
18505 IF ABS(S(M))<C1/Z9 THEN S(M)=C0
18510 NEXT M:MAX=-Z9:MIN=Z9:FOR M=C1 TO NP:Z
      3=V(M)
18515 IF MAX<Z3 THEN MAX=Z3
18520 IF MIN>Z3 THEN MIN=Z3:N=M
18525 NEXT M:MXMN=MAX-MIN:NOW=MIN:IM=N-C1:IF
       IM<C1 THEN IM=NP
18527 IP=N+C1:IF IP>NP THEN IP=C1
18530 IF P(C1)=C0 THEN P(C1)=C1:P(C2)=C0
18535 M=C1:IF P(M)<C0 THEN M=C1
18540 GOTO 18675
18545 REM Calculate intercepts...
18550 IF S(N)=Z9 OR S(N)=C0 THEN Z3=U(N):GOT
      O 18560
18555 Z3=(NOW-T(N))/S(N)
18560 IF S(IM)=Z9 OR S(IM)=C0 THEN Z4=U(IM):
      GOTO 18575
18565 Z4=(NOW-T(IM))/S(IM)
18570 REM Rotate U(),V() to X(),Y():
18575 X1=INT(Z3*CA-NOW*SA+Z5)
18580 IF NOW<>MIN AND P(M)>=C0 THEN 18595
18585 IF P(C0)=C1 THEN IF X1=C2*INT(X1/C2) T
      HEN X1=X1+C1
18590 IF P(C0)=C2 THEN IF X1<>C2*INT(X1/C2)
      THEN X1=X1+C1
18595 Y1=INT(Z3*SA+NOW*CA+Z5)
18600 Y2=INT(Z4*SA+NOW*CA+Z5)
18605 IF SA=C0 THEN X2=Z4*CA
18610 IF SA<>C0 THEN X2=X1+(Y2-Y1)*RA
18615 REM Bar Painting................
18620 IF TYPE=1 THEN PLOT X1,Y1:DRAWTO X2,Y2
      :GOTO 18660
18635 REM Pixel Painting............
18643 IF PB<C1 THEN PR=PB:GOTO 18650
18645 PR=((NOW-MIN)/MXMN)^ABS(PC):IF PC<C0 T
      HEN PR=C1-PR
18650 GOSUB LINEP
18655 REM ..........................
18660 NOW=Y1*CA-X1*SA
18665 REM Increment NOW for next bar
18670 NOW=NOW+ABS(P(M)):M=M+C1:IF P(M)=C0 TH
      EN M=C1
18675 IF NOW<V(IP) THEN 18690
```

```
18680 IF V(IP)=MAX THEN RETURN
18685 N=IP:IP=IP+C1:IF IP>NP THEN IP=C1
18690 IF NOW<V(IM) THEN GOTO 18550
18695 IF V(IM)=MAX THEN RETURN
18700 IM=IM-C1:IF IM<C1 THEN IM=NP
18705 GOTO 18550
18710 REM ============================
18750 REM LINEP Subroutine
18751 REM by... Phil Dunn
18755 REM Draws a line pixel by pixel
18760 REM With a probability to plot
18765 REM or skip each pixel.
18770 REM X1,Y1 = start point
18775 REM X2,Y2 = end point
18780 REM PR = probability to PLOT
18785 REM .{3 SPACES}>=0 and <=1
18790 C0=0:C1=1:C255=255:C230=230:C198=198
18795 C92=92:C97=97:C103=103:C106=106:C120=1
      20
18800 K=X2-X1:L=Y2-Y1:Z3=ABS(K):Z4=ABS(L):BI
      =PR*C255
18805 IF K=C0 AND L=C0 THEN RETURN
18810 IF K<C0 THEN 18840
18815 LINE$(C92,C92)=CHR$(C230):REM INC
18820 LINE$(C97,C97)=CHR$(C0)
18825 LINE$(C103,C103)=CHR$(C1)
18830 LINE$(C106,C106)=CHR$(C230)
18835 GOTO 18860
18840 LINE$(C92,C92)=CHR$(C198):REM DEC
18845 LINE$(C97,C97)=CHR$(C255)
18850 LINE$(C103,C103)=CHR$(C0)
18855 LINE$(C106,C106)=CHR$(C198)
18860 LINE$(C120,C120)=CHR$(C230):IF L<C0 TH
      EN LINE$(C120,C120)=CHR$(C198)
18865 IF Z3=C0 THEN Z3=C1/C255
18870 IF Z4=C0 THEN Z4=C1/C255
18875 IF Z3>=Z4 THEN K=Z3:Y2=Z3/Z4:X2=1:IF Y
      2>C255 THEN Y2=C255
18880 IF Z3<Z4 THEN K=Z4:X2=Z4/Z3:Y2=1:IF X2
      >C255 THEN X2=C255
18883 IF K<1 THEN RETURN
18884 POKE 752,C1
18885 L=USR(ADR(LINE$),K,X1,Y1,X2,Y2,BI)
18890 RETURN
18895 REM ............................
18900 DIM LINE$(136):RESTORE 18910:LINEP=187
      90
18905 FOR K=1 TO 136:READ L:LINE$(K,K)=CHR$(
      L):NEXT K:RETURN
18910 DATA 104,104,133,211,104,133,210,104,1
```

```
      33,86
18915 DATA 104,133,85,104,104,133,84,104,104
      ,133
18920 DATA 209,170,104,104,133,208,168,104,1
      04,133
18925 DATA 207,173,10,210,24,101,207,144,46,
      138
18930 DATA 72,152,72,165,86,72,165,85,72,165
18935 DATA 84,72,162,96,169,11,157,66,3,169
18940 DATA 0,157,72,3,157,73,3,169,1,32
18945 DATA 86,228,104,133,84,104,133,85,104,
      133
18950 DATA 86,104,168,104,170,202,208,19,165
      ,209
18955 DATA 170,230,85,165,85,201,0,208,8,165
18960 DATA 86,201,1,240,30,230,86,169,255,19
      7
18965 DATA 208,240,8,136,208,5,165,208,168,2
      30
18970 DATA 84,198,210,208,162,169,0,197,211,
      240
18975 DATA 4,198,211,240,152,96
```

## PROGRAM 5. Multi-Colored Graphics In Mode 8.

```
100 REM ==============================
110 REM ={9 SPACES}PICTUR8{11 SPACES}=
120 REM ={3 SPACES}An Example  Picture
    {5 SPACES}=
130 REM ={5 SPACES}Program For The
    {7 SPACES}=
140 REM ={4 SPACES}POLY8  Subroutine
    {6 SPACES}=
150 REM ==============================
160 REM ={11 SPACES}by:{13 SPACES}=
170 REM ={8 SPACES}Phil Dunn{10 SPACES}=
180 REM ={6 SPACES}12 Monroe Ave.{7 SPACES}=
190 REM ={3 SPACES}Hicksville, NY 11801
    {4 SPACES}=
200 REM ==============================
250 C0=0:C1=1:C2=2:C3=3:C4=4:C5=5:C10=10:C16
    =16:DETRAP=40000
260 DIM P(C10),X(C16),Y(C16),S(C16),T(C16),U
    (C16),V(C16),EA$(C10),EF$(C10)
270 EA$="AT AREA = "
280 EF$="AT FILL = "
290 POLY8=18400:GRAPHICS 8
300 SETCOLOR C2,C5,C0:SETCOLOR C4,C5,C0
310 SETCOLOR C1,C0,14:COLOR C1
320 REM ==============================
322 REM SET UP LINE SUBROUTINE
325 ? " Reading DATA...":GOSUB 18900
327 REM ==============================
330 ? " CTIA or GTIA chip (1 or 2)";
332 TRAP 330:INPUT CHIP:TRAP DETRAP
334 IF CHIP<>1 AND CHIP<>2 THEN 330
336 REM ==============================
339 AREA=C0
340 AREA=AREA+C1
350 RESTORE 3000+C10*AREA
360 TRAP 730:READ FILL
362 ? "AREA, FILL = ";AREA;", ";FILL
365 IF FILL=999 THEN 1000
370 REM ..............................
380 IF FILL<>-7 THEN 470
385 READ RA
390 IF RA<>C0 AND RA<>C1 THEN ? "COLOR = ";R
    A;"{3 SPACES}";EA$;AREA:STOP
400 TYPE=ABS(FILL)
405 TRAP DETRAP:GOSUB POLY8
410 READ P:IF P=99 THEN 360
420 GOTO 340
460 REM ..............................
```

```
470 READ NP
480 IF NP<C3 OR NP>C16 THEN ? "NP = ";NP;"
    {3 SPACES}";EA$;AREA:STOP
490 FOR I=C1 TO NP:READ X,Y
493 IF X<CO OR Y<CO OR X>319 OR Y>159 THEN ?
     "X,Y=";X;",";Y;"  ";EA$;AREA:STOP
496 X(I)=X:Y(I)=Y:NEXT I
500 REM ...............................
510 IF FILL<CO THEN 400
540 REM ===============================
550 RESTORE 2000+C10*FILL
560 TRAP 740:READ TYPE
570 REM ...............................
580 IF TYPE<>7 THEN 620
585 READ RA
590 IF RA<>CO AND RA<>C1 THEN ? "COLOR = ";R
    A;"{3 SPACES}";EF$;FILL:STOP
600 COLOR RA:GOTO 700
610 REM ...............................
620 IF TYPE<>C2 THEN 640
625 READ PB,PC
630 IF PB<=CO OR PB>C1 THEN ? "PB = ";PB;"
    {3 SPACES}";EF$;FILL:STOP
640 READ RA,PO,P:P(CO)=PO:P(C1)=P:I=C2
650 IF PO<CO OR P=CO THEN ? "PO,P1=";PO;",";
    P1;"{3 SPACES}";EF$;FILL:STOP
652 IF CHIP=C1 THEN 660
654 IF PO=1 THEN P(CO)=2
656 IF PO=2 THEN P(CO)=1
660 READ P:P(I)=P
670 IF P=CO THEN TRAP DETRAP:GOSUB POLY8:GOT
    O 700
680 I=I+C1:GOTO 660
690 REM ...............................
700 READ P:IF P=99 THEN 560
710 GOTO 340
720 REM ===============================
730 ? "ERROR AT AREA DATA = ";AREA:STOP
740 ? "ERROR AT FILL DATA = ";FILL:STOP
750 REM ===============================
1000 ? "{5 SPACES}THE  MOUNTAIN  OF  LIGHT"
1010 ? "  Inspired by  'The Tiger's Fang'"
1020 ? "{4 SPACES}A book by{3 SPACES}Paul Tw
     itchell"
1030 GOTO 1030
1100 REM ==============================
1990 REM FILL Colors & Textures
2000 DATA 7,0,99,1,100,0,1,0,99,7,1,0,HORIZ.
     ERASER
2010 DATA 1,100,0,1,0,0,WHITE=1
```

```
2020 DATA 1,0,1,-2,0,0,DARK BLUE=2
2030 DATA 1,-2,1,3,0,0,RED&GREEN=3
2040 DATA 1,4,1,-9,0,0,NUBBY RED=4
2050 DATA 1,-4,2,-10,0,0,NUBBY RED=5
2060 DATA 1,4,1,-10,0,0,NUBBY BLUE=6
2070 DATA 1,-4,2,-10,0,0,NUBBY BLUE=7
2080 DATA 1,0,2,-1,3,0,0,BAR RED=8
2090 DATA 2,1,-1,100,0,2,2,3,4,5,7,10,13,17,
     0,0,RECEEDING BARS=9
2100 DATA 2,1,1,-0.5,0,1,0,0,LEFT BLEND=10
2110 DATA 2,1,1,0.5,0,1,0,0,RIGHT BLEND=11
2120 DATA 1,-2,1,-2,0,0,LIGHT GREEN=12
2130 DATA 1,2,1,-2,0,0,LIGHT GREEN=13
2140 DATA 2,0.2,1,100,0,2,0,0,TEXTURIZER=14
2990 REM ==============================
3000 REM AREA Datas
3010 DATA -7,1,0,DUMMY=1
3020 DATA -7,1,0,DUMMY=2
3030 DATA 8,4,65,100,65,95,275,95,275,100,0,
     CITY WALLS=3
3040 DATA 2,4,1,158,1,100,318,100,318,158,0,
     OCEAN=4
3050 DATA 9,4,1,158,1,100,318,100,318,158,0,
     REFLEC=5
3060 DATA 0,9,1,158,1,115,25,108,56,113,59,1
     17,59,126,75,130,75,156,80,158,0,ERASE
     LEFT CLIF=6
3070 DATA 12,9,1,158,1,115,25,108,56,113,59,
     117,59,126,75,130,75,156,80,158,0,FILL
     LEFT CLIF=7
3080 DATA 14,9,1,158,1,115,25,108,56,113,59,
     117,59,126,75,130,75,156,80,158,0,TEXTU
     RE LEFT CLIF=8
3090 DATA 0,3,75,130,56,140,75,156,0,ERASE L
     EFT NEAR FACE=9
3100 DATA 3,3,75,130,56,140,75,156,0,FILL LE
     FT NEAR FACE=10
3110 DATA 0,3,59,117,45,122,59,126,0,ERASE L
     EFT FAR FACE=11
3120 DATA 3,3,59,117,45,122,59,126,0,FILL LE
     FT FAR FACE=12
3130 DATA 0,6,205,158,245,135,290,125,305,13
     0,305,155,310,158,0,ERASE RIGHT CLIF=13
3140 DATA 13,6,205,158,245,135,290,125,305,1
     30,305,155,310,158,0,FILL RIGHT CLIF=14
3150 DATA 14,6,205,158,245,135,290,125,305,1
     30,305,155,310,158,0,TEXTURE RIGHT CLIF
     =15
3160 DATA 0,3,305,130,280,145,305,155,0,ERAS
     E RIGHT FACE=16
```

```
3170 DATA 3,3,305,130,280,145,305,155,0,FILL
      RIGHT FACE=17
3180 DATA 1,4,155,125,162,132,168,145,155,14
     1,0,SAIL=18
3190 DATA 1,8,154,141,156,141,159,146,159,14
     8,156,151,154,151,151,148,151,146,0,BOA
     T=19
3200 DATA -7,0,99,-4,10,153,145,153,146,153,
     147,154,146,154,147,156,147,156,148,157
     ,146,157,147,157,148.0
3210 DATA -7,1,0,PAUL & REBEZAR=21
3220 DATA -7,0,99,-5,8,160,144,174,158,158,1
     52,162,158,152,152,146,158,150,144,136,
     158,99,-7,1,0,WAKE=22
3230 DATA 10,10,70,95,95,95,102,87,120,60,13
     0,36,145,18,160,5,165,5,170,10,170,95,0
     ,MTN.LEFT=23
3240 DATA 11,9,170,95,170,10,182,7,192,20,20
     5,32,212,35,225,65,245,95,275,95,0,MTN.
     RIGHT=24
3250 DATA 6,4,145,2,60,2,100,77,105,27,0,LEF
     T RAYS BLUE=25
3260 DATA 7,4,205,5,290,5,235,80,235,30,0,RI
     GHT RAYS BLUE=26
3270 DATA 4,3,150,1,95,85,50,1,0,LEFT RAYS R
     ED=27
3280 DATA 5,3,200,1,240,85,300,1,0,RIGHT RAY
     S RED=28
3290 DATA 999
18000 REM ============================
18005 REM ={4 SPACES}POLY8 Subroutine
      {5 SPACES}=
18010 REM =  Polygon Painting For{3 SPACES}=
18015 REM = BASIC  Graphics Mode 8  =
18020 REM ============================
18025 REM =  by:  Phil Dunn{9 SPACES}=
18030 REM ============================
18035 REM Enter with the value for
18040 REM TYPE= Type of Painting
18045 REM "{3 SPACES}= 1 for Bar Painting
18050 REM "{3 SPACES}= 2 for Pixel Painting
18053 REM "{3 SPACES}= 3 for a Line Boundary
18055 REM "{3 SPACES}= 4 to PLOT X(i),Y(i)
18057 REM "{3 SPACES}= 5 TO PLOT X(i),Y(i),
18058 REM .  and DRAWTO X(i+1),Y(i+1)
18060 REM "{3 SPACES}= 6 to DRAWTO X(i),Y(i)
18065 REM "{3 SPACES}= 7 to COLOR RA
18070 REM ............................
18075 REM For TYPE = 1 to 6,
18080 REM enter with the values...
```

```
18085 REM NP = No. of Vertex Points.
18087 REM .  for i=1 to NP:
18090 REM X(i)= DIM Vertex X values
18095 REM Y(i)= DIM Vertex Y values
18100 REM ...........................
18105 REM For TYPE 1 and 2 Painting
18110 REM also enter values for...
18115 REM RA = Angle X/Y Ratio
18120 REM "  = 0 for Vertical
18125 REM "  = +-1 for +-45 degrees
18130 REM "  = 100 for Horizontal
18135 REM P()= DIM array for spacing
18140 REM .Parity Color-Lock Option:
18145 REM .P(0)= 0 for no parity
18150 REM .P(0)= 1 for odd parity
18155 REM .P(0)= 2 for even parity
18160 REM .for i=1 to something:
18165 REM .ABS(P(i))= Spaces to move
18170 REM .SGN(P(i))=+1, no parity
18175 REM .SGN(P(i))=-1, parity lock
18180 REM .{4 SPACES}P(i) = 0 to end data
18185 REM ...........................
18190 REM For TYPE 2 Pixel Painting
18195 REM also enter values for...
18200 REM PB & PC For Pixel Blending
18205 REM * Set PB<1 for an even
18210 REM . blend of active and
18215 REM . inactive pixels.
18220 REM . PB= the proportion of
18225 REM . active pixels.  PB>0
18230 REM * Set PB>=1 for an uneven
18235 REM . blend across the area.
18240 REM . area.  Then...
18245 REM . If PC>0 then the start
18250 REM .{4 SPACES}color is inactive and
18255 REM .{4 SPACES}the end is active.
18260 REM . If PC<0 then the start
18265 REM .{4 SPACES}color is active and
18270 REM .{4 SPACES}the end is inactive.
18275 REM . When ABS(PC)=1 then the
18280 REM .{4 SPACES}start color phases out
18285 REM .{4 SPACES}evenly to the end.
18290 REM . When ABS(PC)>1 then the
18295 REM .{4 SPACES}start color phases out
18300 REM .{4 SPACES}more slowly.
18305 REM . When ABS(PC)<1 then the
18310 REM .{4 SPACES}start color phases out
18315 REM .{4 SPACES}more rapidly.
18320 REM ...........................
18325 REM For TYPE = 7, to COLOR RA,
18330 REM enter with RA = 0 or 1
18335 REM ............................
18340 REM S()= DIM array used here
18345 REM T()= DIM array used here
18350 REM U()= DIM array used here
18355 REM V()= DIM array used here
18360 REM Variable names used...
18365 REM CO,C1,C2,Z3,Z4,Z5,Z9,
18370 REM X1,Y1,X2,Y2,CA,SA
18375 REM K,L,M,N,IM,IP,MAX,MIN,NOW
18380 REM ============================
18400 CO=0:C1=1:C2=2:Z9=999
18405 IF TYPE=3 THEN PLOT X(C1),Y(C1):FOR N=
      C2 TO NP:DRAWTO X(N),Y(N):NEXT N:DRAWT
      O X(C1),Y(C1):RETURN
18410 IF TYPE=4 THEN FOR N=C1 TO NP:PLOT X(N
      ),Y(N):NEXT N:RETURN
18415 IF TYPE=5 THEN FOR N=C1 TO NP STEP C2:
      PLOT X(N),Y(N):DRAWTO X(N+C1),Y(N+C1):
      NEXT N:RETURN
18420 IF TYPE=6 THEN FOR N=C1 TO NP:DRAWTO X
      (N),Y(N):NEXT N:RETURN
18425 IF TYPE=7 THEN COLOR RA:RETURN
18430 Z5=0.5:IF RA=C2 THEN Z5=0.55
18435 IF RA=-C2 THEN Z5=0.18
18440 Z3=-C1:IF RA<>CO THEN Z3=SGN(RA)
18445 SA=SQR(1/(1+RA^C2))*Z3:IF ABS(SA)<0.2
      THEN SA=CO
18450 CA=SQR(1-SA^C2)
18455 REM Rotate X(),Y() to U(),V():
18460 FOR M=C1 TO NP
18465 U(M)=X(M)*CA+Y(M)*SA
18470 V(M)=-X(M)*SA+Y(M)*CA:NEXT M
18480 FOR M=C1 TO NP:N=M+C1:IF N>NP THEN N=C
      1
18485 REM Calculate slopes S() amd Y axis in
      tercepts T()
18490 IF U(M)=U(N) THEN S(M)=Z9:GOTO 18510
18495 S(M)=(V(N)-V(M))/(U(N)-U(M)):T(M)=V(M)
      -S(M)*U(M)
18500 IF ABS(S(M))>Z9 THEN S(M)=Z9
18505 IF ABS(S(M))<C1/Z9 THEN S(M)=CO
18510 NEXT M:MAX=-Z9:MIN=Z9:FOR M=C1 TO NP:Z
      3=V(M)
18515 IF MAX<Z3 THEN MAX=Z3
18520 IF MIN>Z3 THEN MIN=Z3:N=M
18525 NEXT M:MXMN=MAX-MIN:NOW=MIN:IM=N-C1:IF
       IM<C1 THEN IM=NP
18527 IP=N+C1:IF IP>NP THEN IP=C1
18530 IF P(C1)=CO THEN P(C1)=C1:P(C2)=CO
```

```
18535 M=C1:IF P(M)<CO THEN M=C1
18540 GOTO 18675
18545 REM Calculate intercepts...
18550 IF S(N)=Z9 OR S(N)=CO THEN Z3=U(N):GOT
      O 18560
18555 Z3=(NOW-T(N))/S(N)
18560 IF S(IM)=Z9 OR S(IM)=CO THEN Z4=U(IM):
      GOTO 18575
18565 Z4=(NOW-T(IM))/S(IM)
18570 REM Rotate U(),V() to X(),Y():
18575 X1=INT(Z3*CA-NOW*SA+Z5)
18580 IF NOW<>MIN AND P(M)>=CO THEN 18595
18585 IF P(CO)=C1 THEN IF X1=C2*INT(X1/C2) T
      HEN X1=X1+C1
18590 IF P(CO)=C2 THEN IF X1<>C2*INT(X1/C2)
      THEN X1=X1+C1
18595 Y1=INT(Z3*SA+NOW*CA+Z5)
18600 Y2=INT(Z4*SA+NOW*CA+Z5)
18605 IF SA=CO THEN X2=Z4*CA
18610 IF SA<>CO THEN X2=X1+(Y2-Y1)*RA
18615 REM Bar Painting...............
18620 IF TYPE=1 THEN PLOT X1,Y1:DRAWTO X2,Y2
      :GOTO 18660
18635 REM Pixel Painting............
18643 IF PB<C1 THEN PR=PB:GOTO 18650
18645 PR=((NOW-MIN)/MXMN)^ABS(PC):IF PC<CO T
      HEN PR=C1-PR
18650 GOSUB LINEP
18655 REM ...........................
18660 NOW=Y1*CA-X1*SA
18665 REM Increment NOW for next bar
18670 NOW=NOW+ABS(P(M)):M=M+C1:IF P(M)=CO TH
      EN M=C1
18675 IF NOW<V(IP) THEN 18690
18680 IF V(IP)=MAX THEN RETURN
18685 N=IP:IP=IP+C1:IF IP>NP THEN IP=C1
18690 IF NOW<V(IM) THEN GOTO 18550
18695 IF V(IM)=MAX THEN RETURN
18700 IM=IM-C1:IF IM<C1 THEN IM=NP
18705 GOTO 18550
18710 REM ===========================
18750 REM LINEP Subroutine
18755 REM Draws a line pixel by pixel
18760 REM With a probability to plot
18765 REM or skip each pixel.
18770 REM X1,Y1 = start point
18775 REM X2,Y2 = end point
18780 REM PR = probability to PLOT
18785 REM .{3 SPACES}>=0 and <=1
18790 CO=0:C1=1:C255=255:C230=230:C198=198
```

```
18795 C92=92:C97=97:C103=103:C106=106:C120=1
      20
18800 K=X2-X1:L=Y2-Y1:Z3=ABS(K):Z4=ABS(L):BI
      =PR*C255
18805 IF K=C0 AND L=C0 THEN RETURN
18810 IF K<C0 THEN 18840
18815 LINE$(C92,C92)=CHR$(C230):REM INC
18820 LINE$(C97,C97)=CHR$(C0)
18825 LINE$(C103,C103)=CHR$(C1)
18830 LINE$(C106,C106)=CHR$(C230)
18835 GOTO 18860
18840 LINE$(C92,C92)=CHR$(C198):REM DEC
18845 LINE$(C97,C97)=CHR$(C255)
18850 LINE$(C103,C103)=CHR$(C0)
18855 LINE$(C106,C106)=CHR$(C198)
18860 LINE$(C120,C120)=CHR$(C230):IF L<C0 TH
      EN LINE$(C120,C120)=CHR$(C198)
18865 IF Z3=C0 THEN Z3=C1/C255
18870 IF Z4=C0 THEN Z4=C1/C255
18875 IF Z3>=Z4 THEN K=Z3:Y2=Z3/Z4:X2=1:IF Y
      2>C255 THEN Y2=C255
18880 IF Z3<Z4 THEN K=Z4:X2=Z4/Z3:Y2=1:IF X2
      >C255 THEN X2=C255
18883 IF K<1 THEN RETURN
18884 POKE 752,C1
18885 L=USR(ADR(LINE$),K,X1,Y1,X2,Y2,BI)
18890 RETURN
18895 REM ............................
18900 DIM LINE$(136):RESTORE 18910:LINEP=187
      90
18905 FOR K=1 TO 136:READ L:LINE$(K,K)=CHR$(
      L):NEXT K:RETURN
18910 DATA 104,104,133,211,104,133,210,104,1
      33,86
18915 DATA 104,133,85,104,104,133,84,104,104
      ,133
18920 DATA 209,170,104,104,133,208,168,104,1
      04,133
18925 DATA 207,173,10,210,24,101,207,144,46,
      138
18930 DATA 72,152,72,165,86,72,165,85,72,165
18935 DATA 84,72,162,96,169,11,157,66,3,169
18940 DATA 0,157,72,3,157,73,3,169,1,32
18945 DATA 86,228,104,133,84,104,133,85,104,
      133
18950 DATA 86,104,168,104,170,202,208,19,165
      ,209
18955 DATA 170,230,85,165,85,201,0,208,8,165
18960 DATA 86,201,1,240,30,230,86,169,255,19
      7
```

```
18965 DATA 208,240,8,136,208,5,165,208,168,2
      30
18970 DATA 84,198,210,208,162,169,0,197,211,
      240
18975 DATA 4,198,211,240,152,96
```

# TEXTPLOT Makes A Game

David Plotkin

*The animation capabilities of "Textplot" (see* **COMPUTE!**, *November, 1981, #18) are exploited in Paratroop Attack, a multicolor action game. Requires 24K.*

The machine language subroutine entitled Textplot (**COMPUTE!**, November, 1981, #18) is an excellent tool for animating interesting shapes in Graphics modes 3-7 on the Atari home computer. It is probably the easiest and most straightforward way to effect animation with any kind of speed in these modes. There are really only two things to remember when using Textplot. The first is that the horizontal resolution is only from zero to 19 in Graphics 7, whereas such statements as LOCATE, PLOT, etc., use a resolution from zero to 159. Thus a conversion from the standard Graphics coordinate system to the Textplot coordinate system has to be made. As an example, the point located at Textplot coordinate X=3 is actually at 3*8=24 for the pixel at the far left of the area occupied by the Textplot shape. The horizontal coordinate varies from 24+0=24 to 24+7=31, with pixel 32 at the far left of the area occupied by the Textplot X coordinate equal to four. That is, each Textplot shape is eight pixels wide.

The second thing to remember is that, when using a redefined character set protected by POKEing a lower value of RAMTOP into location 106, the step-back must be 4K, or 16 pages (see line 32000). This is because the Graphics 7 display list, which is located just below RAMTOP, must not cross a 4K boundary, or strange things happen.

One other thing. This program gives you joystick jockeys a break – it uses paddles to ease the wear and tear on the wrists and fingers.

So ... your gun emplacement is under attack by enemy paratroopers. Periodically, a helicopter flies onto the screen and begins dropping paratroopers, who float toward the ground. Should four of the paratroopers get to the ground safely on either side of your gun emplacement, they will blow you up and you will have lost the game. Also, if a paratrooper drops directly onto your gun emplacement, you will be blown up. Your only defense is the high powered laser mounted on your gun emplacement. Its aiming system consists of a target cursor (shaped like a +) which moves about the edge of the screen controlled by Paddle (0). Pressing the red button on the paddle fires the laser, unleashing an energy bolt from the gun to the cursor, destroying everything in its path. Hitting the helicopter causes it to blow up, and hitting the paratrooper causes him to blow up. Hitting the parachute is just as good, since it causes the paratrooper to fall to the ground without his parachute. Also, if one paratrooper lands on another, they are both put out of action.

It takes a while to get used to the paddle aiming system, because there is a delay between moving the paddle and the response of the cursor, so don't turn the paddle too far in trying to get the cursor to move. Unlike with the joystick, however, you can jump the cursor from one edge of the screen to the other in a single flick of the wrist. As your skill increases, the enemy sends better paratroopers against you (they fall faster). A score of 800 points wins the game.

I think you will enjoy this little game as it provides a test of how skillful you are with the paddles on your Atari.

## Documentation

Variables:
CX,CY Coordinates of the target cursor     SC score
H=0, no helicopter; =1, helicopter on screen
HX,HY X and Y coordinates of the helicopter
T(N)=0, paratrooper not on screen; =1, paratrooper on screen
TX(N),TY(N) X and Y coordinates of each paratrooper
P paddle (0)
HP position variable of helicopter
PP position variable of paratrooper
These position variables determine what shapes the helicopter or paratroopers have.

## Program Description

1-4 Initializes subroutines and title page.
5-10 Sets up graphics and dimension arrays.
40-50 Specifies colors and initializes variables.
60-70 Draws terrain and gun emplacement.
75 Puts score on the screen.
80-220 Aims and fires paddle operated laser.
230-280 Tests for hit on the helicopter.

290-340 Tests for hits on the paratroopers.

350-380 Tests for hits on the parachutes.

385-390 Updates score, jumps to end of game on high score.

400-430 Launches new helicopter.

440-470 Advances existing helicopter.

480-510 Launches new paratroopers.

520-580 Advances existing paratroopers and tests for
  540 Landing of paratrooper on ground
  550 Landing of paratrooper on another paratrooper
  560 Landing of paratrooper on the gun emplacement.

525 Increases paratrooper fall rate based on the score

600-650 Subroutine for paratrooper with hole in parachute

660-690 Subroutine for one paratrooper landing on another

700-750 Subroutine for blowing up gun emplacement

800-840 End of game (lost) display

850-870 Erases remaining paratroopers when four have landed.

880-920 Flies helicopter off-screen when four paratroopers have landed.

930-1060 Subroutine to line up paratroopers on the left and destroy the gun emplacement

1070-1190 Subroutine to line paratroopers on the right and destroy the gun emplacement

1200-1280 End of game (won) title and tune

1500-1550 Title page

20000-20430 TEXTPLOT

32000-32200 Redefined character set

## PROGRAM. TEXTPLOT Makes A Game.

```
1 GOSUB 32000:CLR
2 GOSUB 20000
3 GOSUB 1500
5 DIM TX(25),TY(25),T(25)
10 GRAPHICS 23:POKE 752,1:SC=0:POKE 756,PEEK
   (106)
40 SETCOLOR 4,9,2:SETCOLOR 2,12,6
50 P=0:FOR N=1 TO 25:T(N)=0:TX(N)=0:TY(N)=0:
   NEXT N:N1=0:N2=0:H=0:HX=0:HY=0:CX=0:CY=0
60 COLOR 3:FOR W=80 TO 92:PLOT 0,W:DRAWTO 15
   9,W:NEXT W
70 COLOR 1:FOR Y=72 TO 79:PLOT 75,Y:DRAWTO 8
   5,Y:NEXT Y:D=USR(1536,9,1,10,66)
75 D=USR(1536,48,2,1,82):D=USR(1536,48,2,2,8
   2):D=USR(1536,48,2,3,82)
80 FOR M=1 TO 2:D=USR(1536,32,2,CX,CY)
90 P=PADDLE(0):IF P<165 THEN GOTO 110
100 CY=P-164:CX=0:GOTO 140
110 IF P<65 THEN GOTO 130
120 CX=INT((164-P)/5):CY=2:GOTO 140
130 CX=19:CY=64-P
140 COLOR 2:D=USR(1536,8,2,CX,CY):NEXT M
150 IF PTRIG(0)=1 THEN GOTO 230
160 IF P>190 THEN D=USR(1536,9,1,10,66):GOTO
     210
170 IF P>154 THEN D=USR(1536,10,1,10,66):GOT
    O 210
180 IF P>80 THEN D=USR(1536,11,1,10,66):GOTO
     210
190 IF P>44 THEN D=USR(1536,12,1,10,66):GOTO
     210
200 D=USR(1536,13,1,10,66)
210 COLOR 2:PLOT 80,65:DRAWTO 8*CX,CY:COLOR
    0:PLOT 80,65:DRAWTO 8*CX,CY
220 FOR X=1 TO 10:SOUND 0,30,10,8:NEXT X:SOU
    ND 0,0,0,0
230 IF H=0 THEN GOTO 290
240 FOR M=2 TO 6:LOCATE HX*8+M,HY+3,Z:IF Z<>
    0 THEN GOTO 280
250 SOUND 1,0,0,0:SOUND 0,30,8,8:D=USR(1536,
    14,2,HX,HY):SC=SC+10
260 FOR Q=1 TO 30:NEXT Q
270 SOUND 0,0,0,0:D=USR(1536,32,2,HX,HY):H=0
    :HX=0:HY=0:GOTO 290
280 NEXT M
290 FOR M=1 TO 5:IF T(M)=0 THEN GOTO 340
300 LOCATE TX(M)*8+3,TY(M)+4,Z:LOCATE TX(M)*
    8+4,TY(M)+4,Z1:LOCATE TX(M)*8+4,TY(M)+3,
    Z2:LOCATE TX(M)*8+2,TY(M)+4,Z3
```

```
305 IF Z<>0 AND Z1<>0 AND Z2<>0 AND Z3<>0 TH
    EN GOTO 340
310 SOUND 0,15,8,8:D=USR(1536,14,2,TX(M),TY(
    M)):SC=SC+5
320 FOR Q=1 TO 20:NEXT Q
330 SOUND 0,0,0,0:D=USR(1536,32,2,TX(M),TY(M
    )):T(M)=0:TX(M)=0:TY(M)=0
340 NEXT M
350 FOR M=1 TO 5:IF T(M)=0 THEN GOTO 380
360 FOR N=2 TO 5:LOCATE TX(M)*8+N,TY(M),Z:IF
     Z=0 THEN GOSUB 600:SC=SC+5:GOTO 380
370 NEXT N
380 NEXT M
385 A=INT(SC/100):B=INT(SC/10)-A*10:C=SC-100
    *A-10*B:A=A+48:B=B+48:C=C+48
386 D=USR(1536,A,2,1,82):D=USR(1536,B,2,2,82
    ):D=USR(1536,C,2,3,82)
390 IF SC>800 THEN GOTO 1200
400 IF H<>0 THEN GOTO 440
410 W=INT(RND(0)*3+1):IF W>1 THEN GOTO 520
420 H=1:SOUND 1,20,8,4:HX=1:HY=INT(RND(0)*30
    +10):HP=6
430 D=USR(1536,6,1,HX,HY):GOTO 480
440 D=USR(1536,32,1,HX,HY):HX=HX+1:IF HX>18
    THEN H=0:SOUND 1,0,0,0:GOTO 520
450 IF HP=6 THEN HP=7:GOTO 470
460 HP=6
470 D=USR(1536,HP,1,HX,HY)
480 FOR N=1 TO 5:IF T(N)<>0 THEN GOTO 510
490 W=INT(RND(0)*4+1):IF W=1 AND HX>=2 THEN
    T(N)=1:TX(N)=HX-1:TY(N)=HY:D=USR(1536,0,
    3,TX(N),TY(N))
500 GOTO 520
510 NEXT N
520 FOR N=1 TO 5:IF T(N)=0 THEN GOTO 580
525 J=INT(SC/100)+1:IF TY(N)+J>66 THEN J=67-
    TY(N):IF J<1 THEN J=1
530 D=USR(1536,32,3,TX(N),TY(N)):TY(N)=TY(N)
    +J
540 IF TY(N)<>72 THEN GOTO 550
545 D=USR(1536,1,3,TX(N),TY(N)):T(N)=0:IF TX
    (N)<10 THEN N1=N1+1:GOTO 547
546 N2=N2+1
547 IF N1=4 OR N2=4 THEN GOTO 850
548 GOTO 580
550 IF TY(N)=67 THEN LOCATE TX(N)*8+3,76,Z:I
    F Z=3 THEN GOSUB 660:GOTO 580
560 IF TY(N)>=58 AND TX(N)=10 THEN GOTO 700
570 D=USR(1536,0,3,TX(N),TY(N))
580 NEXT N
590 GOTO 80
```

```
600 D=USR(1536,1,3,TX(M),TY(M))
610 FOR Q=TY(M) TO 72 STEP 2:D=USR(1536,1,3,
    TX(M),Q)
615 IF Q<>67 AND Q<>68 THEN GOTO 620
616 LOCATE TX(M)*8+3,76,Z:IF Z<>3 THEN GOTO
    620
617 IF TX(M)<10 THEN N1=N1-1
618 N2=N2-1
620 D=USR(1536,32,3,TX(M),Q):SOUND 2,90-Q,10
    ,8
630 NEXT Q:D=USR(1536,4,3,TX(M),72):T(M)=0:T
    Y(M)=0
640 FOR Q=1 TO 30:SOUND 2,30,8,8:NEXT Q:SOUN
    D 2,0,0,0
650 RETURN
660 D=USR(1536,4,3,TX(N),72):SOUND 0,30,8,8:
    D=USR(1536,14,2,TX(N),TY(N))
670 FOR Q=1 TO 30:NEXT Q:D=USR(1536,32,2,TX(
    N),TY(N)):SOUND 0,0,0,0
680 T(N)=0:IF TX(N)<10 THEN N1=N1-1:RETURN
690 N2=N2-1:RETURN
700 D=USR(1536,1,3,10,60):FOR Q=1 TO 50:NEXT
     Q
710 FOR Q=54 TO 80 STEP 4:FOR Q1=9 TO 11
720 SOUND 0,Q,8,8:SOUND 1,Q1,8,8:D=USR(1536,
    14,2,Q1,Q)
730 FOR M=1 TO 10:NEXT M:D=USR(1536,32,2,Q1,
    Q)
740 NEXT Q1:NEXT Q:SOUND 0,0,0,0:SOUND 1,0,0
    ,0
750 FOR Q=1 TO 80:NEXT Q
800 GRAPHICS 1+16:PRINT #6;"YOU LOST!! YOUR"
    :PRINT #6;"GUN WAS DESTROYED!"
810 PRINT #6;"FINAL SCORE  ";SC:PRINT #6;"PO
    INTS. PRESS FIRE"
820 PRINT #6;"TO PLAY AGAIN"
830 IF PTRIG(0)=1 THEN GOTO 830
840 GOTO 10
850 FOR N=1 TO 10:IF T(N)=0 THEN GOTO 870
860 D=USR(1536,32,3,TX(N),TY(N))
870 NEXT N
880 IF H=0 THEN GOTO 930
890 FOR N=HX TO 19:IF HP=6 THEN HP=7:GOTO 91
    0
900 HP=7
910 D=USR(1536,HP,1,N,HY):D=USR(1536,32,1,N,
    HY)
920 NEXT N
930 SOUND 1,0,0,0:IF N2=4 THEN GOTO 1070
940 FOR N=1 TO 4:FOR Q=N TO 10:TX=(10-Q)*8+3
950 LOCATE TX,76,Z:IF Z=3 THEN PP=2:GOTO 970
960 NEXT Q
```

```
970 FOR TT=10-Q TO 10-N:D=USR(1536,PP,3,TT,7
    2):FOR W=1 TO 50:NEXT W
980 IF PP=2 THEN PP=3:GOTO 1000
990 PP=2
1000 D=USR(1536,32,3,TT,72):NEXT TT:D=USR(15
     36,1,3,TT-1,72)
1010 NEXT N
1020 D=USR(1536,5,3,9,72):D=USR(1536,32,3,6,
     72):FOR W=1 TO 50:NEXT W
1030 D=USR(1536,5,3,9,64):D=USR(1536,32,3,7,
     72):FOR W=1 TO 50:NEXT W
1040 D=USR(1536,5,3,9,56):D=USR(1536,32,3,8,
     72):FOR W=1 TO 50:NEXT W
1050 D=USR(1536,2,3,9,48):FOR W=1 TO 50:NEXT
      W
1060 D=USR(1536,32,3,9,48):GOTO 700
1070 FOR N=1 TO 4:FOR Q=N TO 9:TX=(10+Q)*8+3
1080 LOCATE TX,76,Z:IF Z=3 THEN PP=2:GOTO 11
     00
1090 NEXT Q
1100 FOR TT=10+Q TO 10+N STEP -1:D=USR(1536,
     PP,3,TT,72):FOR W=1 TO 50:NEXT W
1110 IF PP=2 THEN PP=3:GOTO 1130
1120 PP=2
1130 D=USR(1536,32,3,TT,72):NEXT TT:D=USR(15
     36,1,3,TT+1,72)
1140 NEXT N
1150 D=USR(1536,5,3,11,72):D=USR(1536,32,3,1
     4,72):FOR W=1 TO 50:NEXT W
1160 D=USR(1536,5,3,11,64):D=USR(1536,32,3,1
     3,72):FOR W=1 TO 50:NEXT W
1170 D=USR(1536,5,3,11,56):D=USR(1536,32,3,1
     2,72):FOR W=1 TO 50:NEXT W
1180 D=USR(1536,2,3,11,48):FOR W=1 TO 50:NEX
     T W
1190 D=USR(1536,32,3,11,48):GOTO 700
1200 GRAPHICS 1+16:PRINT #6;"VERY GOOD!! YOU
     ":PRINT #6;"WON THIS ROUND!"
1210 SOUND 0,40,10,8:GOSUB 1280
1220 SOUND 0,30,10,8:GOSUB 1280
1230 SOUND 0,24,10,8:GOSUB 1280
1240 SOUND 0,20,10,8:GOSUB 1280
1250 SOUND 0,24,10,8:GOSUB 1280
1260 SOUND 0,20,10,8:GOSUB 1280:GOSUB 1280:G
     OSUB 1280
1270 SOUND 0,0,0,0:GOTO 810
1280 FOR W=1 TO 45:NEXT W:RETURN
1500 GRAPHICS 1+16:FOR Q=1 TO 7:PRINT #6:NEX
     T Q:PRINT #6;"*PARATROOP ATTACK*":PRINT
      #6;"by DAVID PLOTKIN"
1510 FOR Q=1 TO 15:SETCOLOR 2,Q,5
1520 FOR W=1 TO 20:SOUND 0,56,10,8:SOUND 1,5
```

```
     6,6,4:NEXT W
1530 FOR W=1 TO 20:SOUND 0,78,10,8:SOUND 1,7
     8,6,4:NEXT W
1540 NEXT Q:SOUND 0,0,0,0:SOUND 1,0,0,0
1550 RETURN
20000 ML=1536:FOR I=0 TO 252:READ A:POKE ML+
      I,A:NEXT I:RETURN
20010 DATA 104,240,10,201,4,240
20020 DATA 11,170,104,104,202,208
20030 DATA 251,169,253,76,164,246
20040 DATA 104,133,195,104,201,128
20050 DATA 144,4,41,127,198,195
20060 DATA 170,141,250,6,224,96
20070 DATA 176,15,169,64,224,32
20080 DATA 144,2,169,224,24,109
20090 DATA 250,6,141,250,6,104
20100 DATA 104,141,251,6,104,104
20110 DATA 141,252,6,14,252,6
20120 DATA 104,104,141,253,6,133
20130 DATA 186,166,87,169,10,224
20140 DATA 3,240,8,169,20,224
20150 DATA 5,240,2,169,40,133
20160 DATA 207,133,187,165,88,133
20170 DATA 203,165,89,133,204,32
20180 DATA 228,6,24,173,252,6
20190 DATA 101,203,133,203,144,2
20200 DATA 230,204,24,165,203,101
20210 DATA 212,133,203,165,204,101
20220 DATA 213,133,204,173,250,6
20230 DATA 133,187,169,8,133,186
20240 DATA 32,228,6,165,212,133
20250 DATA 205,173,244,2,101,213
20260 DATA 133,206,160,0,162,8
20270 DATA 169,0,133,208,133,209
20280 DATA 177,205,69,195,72,104
20290 DATA 10,72,144,8,24,173
20300 DATA 251,6,5,208,133,208
20310 DATA 224,1,240,8,6,208
20320 DATA 38,209,6,208,38,209
20330 DATA 202,208,228,104,152,72
20340 DATA 160,0,165,209,145,203
20350 DATA 200,165,208,145,203,104
20360 DATA 168,24,165,203,101,207
20370 DATA 133,203,144,2,230,204
20380 DATA 200,192,8,208,183,96
20390 DATA 169,0,133,212,162,8
20400 DATA 70,186,144,3,24,101
20410 DATA 187,106,102,212,202,208
20420 DATA 243,133,213,96,0,1
20430 DATA 28
32000 POKE 106,PEEK(106)-16:GRAPHICS 0:START
```

```
      =(PEEK(106))*256:POKE 756,START/256:PO
      KE 752,1
32010 ? "INITIALIZING...TAKES ABOUT 40 SECON
      DS"
32020 FOR Z=0 TO 1023:POKE START+Z,PEEK(5734
      4+Z):NEXT Z:RESTORE 32100
32030 READ X:IF X=-1 THEN RESTORE :RETURN
32040 FOR Y=0 TO 7:READ Z:POKE X+Y+START,Z:N
      EXT Y:GOTO 32030
32100 DATA 512,126,195,129,90,60,24,36,66
32101 DATA 520,0,0,24,24,126,24,36,66
32102 DATA 528,0,0,24,24,56,24,16,24
32103 DATA 536,0,0,24,24,28,24,36,66
32104 DATA 544,0,0,0,0,0,0,90,126
32105 DATA 552,66,66,90,90,126,24,36,66
32106 DATA 560,31,4,142,126,14,4,14,0
32107 DATA 568,0,4,142,126,14,4,14,0
32108 DATA 576,0,0,0,16,56,16,0,0
32109 DATA 584,0,0,248,24,60,60,126,126
32110 DATA 592,128,64,32,24,60,60,126,126
32111 DATA 600,16,16,16,24,60,60,126,126
32112 DATA 608,1,2,4,24,60,60,126,126
32113 DATA 616,0,0,31,24,60,60,126,126
32114 DATA 624,73,42,20,119,20,42,73,8
32115 DATA -1
```

# Fun With Scrolling

David Plotkin

*While this article doesn't tackle the finer points of pixel scrolling, it does present several useful BASIC routines to help you learn more about "coarse" scrolling ... the ability to move lines of graphics vertically and horizontally.*

Many of the graphic capabilities of the Atari home computers have been documented in **COMPUTE!** Magazine: alternate character sets ("Superfont")[1]; use of characters in graphics modes ("Textplot")[2]; and several articles on Player/Missile graphics, including some excellent machine language subroutines. Notably absent has been one of the more spectacular abilities of the Atari – scrolling. For those who don't know, scrolling is the movement of the text or graphics on the screen in whole or in part. The arcade games Scramble and Defender use scrolling. And few of you avid gamesters have not seen Greg Christensen's "Caverns of Mars," a game that is so good that Atari itself is marketing it as "official" Atari. In the pages that follow, I will tell you how to scroll, and provide a program for a game which not only scrolls but includes some other tricks with P/M graphics.

First, though, a few words about the types of scrolling. Horizontal scrolling scrolls left and right; vertical scrolling scrolls up and down. Two subsets are coarse and fine scrolling, both applicable to horizontal and vertical scrolling. Coarse and fine scrolling can be combined to produce combined scrolling (tricky, huh?), which is the type of scrolling that games such as "Caverns of Mars" use.

What does all this mean? Well ... *coarse* scrolling is movement of text or graphics in increments of one letter or one row or one column. Thus, graphics mode 3 has 24 rows from the top to the bottom of the screen, and 24 coarse vertical scrolls will move all the picture on the original screen off the screen. *Fine* scrolling allows for scrolling in the pixel elements of coarse scrolling, so the motion appears smoother. For example, in graphics mode 3, each coarse scroll is broken into eight finer scrolls. Fine scrolling by itself can only move screen data a total of one row or column, so the combination of coarse and fine scrolling is used to produce smooth motion over as many screens of data as required. Unfortunately (although combined scrolling is not particularly difficult or complicated), the necessary transition from fine to coarse scrolling and back which occurs during combined scrolling must happen very fast, too fast for BASIC. Otherwise, there will be some distracting displays on the screen.

A machine language routine for combined scrolling would probably do the screen manipulations during a vertical blank, which occurs 60 times a second when the TV electron gun has finished drawing the picture and is returning to the top of the tube to draw the next screen. Thus, the distracting graphics which occur when using combined scrolling in BASIC would not be seen.

The balance of this article will deal only with coarse scrolling, which is all you need for most applications. The Tricky Tutorial #2 by Santa Cruz Educational Software presents a machine language routine (for inclusion in a BASIC program – you don't need an assembler cartridge) for vertical combined scrolling which works very well. I will not reproduce it here, but I urge readers interested in learning combined scrolling to obtain the program and try the examples. The Tricky Tutorial #2 and this article will give you the necessary tools for some really great graphics displays.

Atari home computers keep the address of the Display List at memory locations 560 and 561 (PEEK(560) + 256*PEEK(561)). The Display List is a set of instructions in memory that the computer uses to find out what to put on the screen. Every time you use a graphics # command, the computer creates a Display List somewhere in memory and puts the address at locations 560 and 561. The fifth and sixth numbers in the Display List are the address of screen memory – the first byte to be displayed on the screen. If you adjust the value of the number stored here, all the data on the screen will move – it will scroll!

So plug in your joystick and punch in the following program:

```
10 GRAPHICS 3:COLOR 1:PLOT 0,0:DRAWTO 40,20
20 DL=PEEK(560)+256*PEEK(561)
30 DL4=DL+4:DL5=DL+5:NUML=PEEK(DL4):NUMH=PEE
   K(DL5)
40 ST=STICK(0)
50 IF ST=11 THEN NUML=NUML+1
60 IF ST=7 THEN NUML=NUML-1
70 IF NUML<0 THEN GOTO 110
80 IF NUML>256 THEN GOTO 140
90 NUML=NUML-256:NUMH=NUMH+1
100 GOTO 120
110 NUML=NUML+256:NUMH=NUMH-1
120 IF NUMH<0 THEN GOTO 40
```

```
130 IF NUMH>255 THEN GOTO 40
140 POKE DL4,NUML:POKE DL5,NUMH
150 GOTO 40
```

Line 20 gets the address of the Display List, while line 30 establishes the variable for the fifth and sixth numbers on the Display List (DL4 is the fifth number). The loop from 40 to 60 changes the variable corresponding to the low part of the screen memory based on the position of the joystick. Lines 70 to 110 adjust the low part (NUML) and high part (NUMH) of screen memory to keep the values from going outside "legal" values. Thus, if NUML gets above 255, then you subtract 256 from it and add one to NUMH. And if NUML gets below zero, then you add 256 to it and subtract one from NUMH. Lines 120 and 130 keep the high part of screen memory from going outside of its limits. Once you reach NUMH = 0 or NUMH = 256 that's as far as you can go. Finally, line 140 POKEs the adjusted values of the address of screen memory and goes back to start again.

A few things you will notice about this program: as you scroll right with your joystick, the picture on the screen disappears off the left edge of your screen and reappears from the right edge. After ten horizontal scrolls, the line is once more on the screen, but now is displaced up one line. This is because in graphics 3, there are ten bytes per line of memory. Below is a chart of graphics modes versus some useful quantities.

| GRAPHICS MODE | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Rows per screen | 24 | 24 | 12 | 24 | 48 | 48 | 96 | 96 | 192 |
| Bytes per row | 40 | 20 | 20 | 10 | 10 | 20 | 20 | 40 | 40 |

Every time you scroll a horizontal distance equal to the number of bytes per line (ten times in graphics 3, 40 times in graphics 7, etc.) you have scrolled vertically by one line. It looks, then, like horizontal and vertical scrolling are really the same thing. Of course, if you have ever played "Eastern Front" by Chris Crawford, you know it's possible to scroll horizontally without the same data coming back on the screen. As far as I can tell, to do that you have to modify the Display List to get the Atari to think that your TV screen is really more than one screen wide.

For the rest of this article, we'll stick to vertical scrolling. Let's change a few lines of the previous program:

```
50 NUML=NUML+10*(ST=4)-10*(ST=13):NUMH=NUMH+
   (NUML>255)-(NUML<0)
60 NUML=NUML+256*(NUML<0)-256*(NUML>255)
```

and delete lines 70-110. Run the program and push your joystick forward and back. The new line 50 increments the value of NUML by 10, depending on the joystick position. It also takes care of incrementing the value of NUMH. Line 60 is responsible for adjusting the value of NUML if it goes outside its limits. Note that the expressions in parentheses are equal to one if the expression is true, and zero if the expression is false. This is much more efficient than the first program. By scrolling up and down in increments of ten (the number of bytes per line), you can achieve vertical scrolling without any horizontal movement.

So far you have not done anything really useful for making graphics displays. You can scroll down, leaving you with an empty screen, or up, which shortly produces a brightly colored jumble on the screen. What has happened is that you are displaying an area of RAM which has data in it. What you need is an empty protected area of RAM to put your own pictures into. To get the memory you need, we'll use a standard Atari trick. Location 106 holds the number of pages available in RAM. You can find this number, which will vary depending on how much memory you have installed in your computer, by doing a PEEK(106). If you then POKE a number into location 106 which is less than the original number, the difference is now protected and won't be used by the computer because it doesn't know that it is there. See the program following to see exactly how to code these commands. Now set up a loop to read zeroes into your protected memory and it will be blank.

The key to scrolling is defining the multi-screen picture to scroll across. I've already told you about locations DL4 and DL5, which hold the address of the beginning of screen memory. To understand how to set up your own pictures, there are two more memory locations you need to know about: 88 and 89. These memory locations hold the address of the *start of write memory* – the memory location where the computer is to execute commands from the keyboard or from a running program. The reason that you see the results of keyboard or program inputs on the screen is that, normally, the address of Display Memory and the address of Write Memory are the same. But they don't have to be. If you change the address of Write Memory and then execute some PLOTs and DRAWTOs, you will see nothing on the screen. However, when you change the Display Memory to match the Write Memory, the picture you've drawn will flash onto the screen. This is a technique known as page flipping.

To utilize this technique, change the Write Memory to write into your protected area of memory and draw the screen as you normally would. Then change the address of the Write Memory to be one screen away from where it was, and draw another screen. You can keep this up as long as you like, generating as many screens edge-to-edge as your memory allows. Then place your Display Memory at one end or the other (or in the middle if you like), and scroll away. The screens will run together if you've done it right, so you won't see the start of one screen and the end of another. And what is the memory length of one screen, for incrementing the Write Memory? Refer back to my chart. If you take the number of rows per screen and multiply by the number of bytes per row, you'll get the number of bytes per screen to move the Write Memory. In Graphics 3, for example, one screen is 24*10=240 bytes. Note that, in general, you will have to change both the high and low parts of Write Memory (locations 89 and 88, respectively) to get a total move of 240 bytes. The math is the same as that for the Display Memory since:

Write Memory =PEEK(88)+256*PEEK(89)

and the low part and high part of Write Memory have the same limits as those for the Display Memory. Also notice that each Graphics 7 screen uses up almost 4K of memory! The subroutine which begins at location 4900 sets up the cavern for the game. I've used the same screen pattern several times to generate a long cavern from just a few different screens. Each time one screen is drawn, I change the Write Memory by 240 bytes and draw the next screen. The POKE 559,0 just turns off the screen and processor to speed up drawing the cavern. POKE 559,34 turns them back on again.

P/M graphics is pretty much ideal for the user-controlled shapes when using scrolling, since it is displayed through a separate system from normal graphics and, as a result, doesn't move when scrolling or page flipping. The excellent machine language routine VBLANK PM[3] was used for moving the space ship, missile, and pterodactyls in the program at the end of this article. I will refer you to that issue to familiarize yourself with VBLANK PM. One thing that it does not seem to be able to do is to change the shapes of the players. Thus, there is no way to blow up the ship or missile, or make the shapes appear different when moving left or right. This is *not* criticism. The authors were trying to make calls of the program from BASIC unnecessary, and they've done an excellent job. Nonetheless, I needed to change the shapes, and there's a fast and easy way to do that.

Remember "Extending Player Missile Graphics"[4]? In that article, a machine language program was presented. Its strong suit was that it could change the shape of players very quickly. It was called by a command of the form:

A=USR(XXX,PMBASE+FM+Y, MEM)

where XXX=address of the machine language routine

FM=First memory location of the player you want to change (512 for pl.0, double line resolution, 1024 for pl.0, single line resolution, etc.)

Y=horizontal coordinate of player

MEM=memory address of shape to change player to.

I read this routine into memory between 256 and 511, which are empty and protected, since page 6 (1536-1792) is used for VBLANK PM. The data for player shapes was put into the empty P/M memory from PMBASE to the beginning of Player 0. Once done, every time I needed to change a player shape, I used a call to Mr. Stoltman's routine, and voilà!

And so I present "Cavern Battle." Your spaceship hangs poised over a deep cavern, at the bottom of which lies a big blue box full of treasure. The object: get to the bottom of the cavern, retrieve the treasure, and get out again, all the while avoiding or destroying the pterodactyls trying to keep you from reaching the treasure. The pterodactyls can move through the walls of the cavern, but don't you try it or you'll blow up! To move your spaceship, use joystick 0, and press the red button to fire your missile. You can only have one missile in flight at a time, so don't waste them. If the pterodactyls catch you, you'll probably blow up, so be careful. You have several things in your favor – your ship is faster and there are only so many prehistoric birds per cavern. But they are tenacious and come unerringly for you. When you reach the bottom of the cavern, hover over the treasure and you'll hear a tune announcing recovery of the treasure. Head for the surface and if you make it, you'll hear the little tune again. If you want to play again, just start down the cavern again. But this time there will be more pterodactyls. One note of warning: When you first RUN the program, the screen will go blank for about 45 seconds while the caverns are drawn. This is normal. Good luck and good hunting!!

## LIST Of VARIABLES

**NUML**: Low part of screen memory

**NUMH**: High part of screen memory

**SF**: Ship flag position-equal to 1 when ship is facing right, equal to -1 when ship is facing left

**MF**: Missile position flag – works like SF except also equal to zero when missile is exploded or not launched

**P2,P3**: Position flags for Players 2 and 3. Work like MF

**X0,X1,X2,X3**: X coordinates of Players

**Y0,Y1,Y2,Y3**: Y coordinates of Players

**PMBASE**: Memory location of beginning of PM graphics

**T2,T3**: Temporary variables for remembering values of P2 and P3

**M,MON**: Keep track of number of Pterodactyls.

**BOT**: =0 when ship has not reached bottom of cavern
=1 when ship has reached bottom of cavern

## PROGRAM DESCRIPTION

**Line 40**: Sets high and low part of screen memory variables.

**Lines 50-180**: Moves ship, scrolls background, keeps Pterodactyls and missile from moving off the screen, and makes Pterodactyls move up and down when background scrolls.

**Lines 150 and 160**: Changes ship direction.

**Line 155**: Detects arrival of ship at bottom of cavern.

**Line 156**: Detects arrival of ship at top of cavern after recovery of treasure.

**Lines 162, 165**: Line 165 detects a hit on the ship. Line 162 clears the register which detects a collision between the ship and the background. The need to do this arises from the fact that occasionally the background flashes when you are scrolling. This will register as a collision unless the collision register is cleared. This sequence also will give an element of randomness to the detection of a collision, since a collision will only be registered if a resetting of the collision register occurs between the execution of line 162 and 165. If you don't like this randomness, add a small waiting loop to the end of line 162. Since you only have to delay 1/60 of a second to allow the collision register to reset, only a small loop is required (resetting occurs during a Vertical Blank).

**Lines 170 and 180**: Resets ship position flags.

**Lines 200-215**: Launches missile.

**Line 220**: Advances missile and detects for a collision.

**Lines 230-240:** Explodes missile.

**Lines 250-260**: Determines if a Pterodactyl is hit – if it is, moves it off the screen.

**Lines 310-350**: Launches new Pterodactyls.

**Lines 380-440**: Changes bird positions and advances them.

**Lines 500-560**: Resets ship to top of cavern when it is destroyed. Starts game over again when five ships are lost.

**Lines 600-640:** Plays tune for arrival at the treasure and at top of cavern.

**Lines 1000-2100**: Initializes VBLANK PM and reads it into memory.

**Line 1075**: Reads data for various player shapes into memory.

**Line 4900**: Steps back top of memory.

**Line 4910**: Clears protected memory.

**Lines 4920-4965**: Changes Write Memory and draws each screen.

**Lines 4970-4980**: Defines Display Memory address variables.

**Lines 5100-5500**: Data for drawing screens.

## References:

[1]"Superfont," **COMPUTE!**, January, 1982, #20
[2]"Textplot," **COMPUTE!**, November, 1981, #18
[3]"P/M Graphics Made Easy," **COMPUTE!**, February, 1982, #21
[4]"Extending Player Missile Graphics," **COMPUTE!**, October, 1981, #17

**PROGRAM. Fun With Scrolling.**

```
20 GOSUB 4900
30 GOSUB 1000
40 NUML=PEEK(DL4):NUMH=PEEK(DL5):SF=1:MF=0:N
   S=5:P2=-1:P3=-1:M=25:BOT=0:MON=M
50 IF STICK(0)=15 THEN GOTO 50
60 ST=STICK(0):IF ST=15 AND STRIG(0)=1 AND P
   EEK(53260)=0 THEN GOTO 215
80 NUML=NUML+10*(ST=14)-10*(ST=13):NUMH=NUMH
   +(NUML>255)-(NUML<0)
90 NUML=NUML+256*(NUML<0)-256*(NUML>255)
100 IF NUMH=RT+14 AND NUML>48 THEN NUML=48:G
    OTO 60
110 IF NUMH=RT+1 AND NUML<136 THEN NUML=136:
    GOTO 60
115 IF P2=0 THEN GOTO 126
120 Y2=Y2-8*(ST=14)+8*(ST=13):IF Y2<32 THEN
    Y2=32
125 IF Y2>224 THEN Y2=224
126 IF P3=0 THEN GOTO 139
130 Y3=Y3-8*(ST=14)+8*(ST=13):IF Y3<32 THEN
    Y3=32
135 IF Y3>224 THEN Y3=224
139 IF MF=0 THEN GOTO 145
140 Y1=Y1-8*(ST=14)+8*(ST=13):IF Y1<32 THEN
    Y1=32
143 IF Y1>224 THEN Y1=224
145 POKE PLY+2,Y2:POKE PLY+3,Y3:POKE DL4,NUM
    L:POKE DL5,NUMH:IF MF<>0 THEN POKE PLY+1
    ,Y1:POKE 53278,1
150 X0=X0+8*(ST=7)-8*(ST=11):POKE PLX,X0:IF
    ST=7 AND SF=-1 THEN D=USR(260,PMBASE+102
    4+Y0,PMBASE+1)
155 IF NUML=194 AND NUMH=RT+13 AND X0=124 TH
    EN BOT=1:GOSUB 600
156 IF NUML=136 AND NUMH=RT+1 AND BOT=1 THEN
     BOT=0:M=MON+5:MON=M:GOTO 600
160 IF ST=11 AND SF=1 THEN D=USR(260,PMBASE+
    1024+Y0,PMBASE+9)
162 POKE 53278,1
165 IF PEEK(53260)<>0 OR PEEK(53252)<>0 THEN
     GOTO 500
170 IF ST=7 THEN SF=1
180 IF ST=11 THEN SF=-1
190 IF STRIG(0)=1 OR MF=1 OR MF=-1 THEN GOTO
     215
200 MF=SF:X1=X0+8*(MF=1)-8*(MF=-1):Y1=Y0:POK
    E PLX+1,X1:POKE PLY+1,Y1
210 D=USR(260,PMBASE+1280+Y1,PMBASE+17*(MF=1
    )+25*(MF=-1)):IF PEEK(53253)<>0 OR PEEK(
    53261)<>0 THEN GOTO 230
215 IF MF=0 THEN GOTO 300
220 IF PEEK(53253)=0 AND PEEK(53261)=0 THEN
    X1=X1+4*(MF=1)-4*(MF=-1):POKE PLX+1,X1:G
    OTO 300
230 D=USR(260,PMBASE+1280+Y1,PMBASE+33):SOUN
    D 0,100,8,8
235 FOR W=1 TO 50:NEXT W:D=USR(260,PMBASE+12
    80+Y1,PMBASE+41):MF=0:SOUND 0,0,0,0
240 IF PEEK(53261)=0 THEN GOTO 270
250 IF PEEK(53262)<>0 THEN X2=0:Y2=0:POKE PL
    X+2,X2:POKE PLY+2,Y2:T2=P2:P2=0:M=M-1
260 IF PEEK(53263)<>0 THEN X3=0:Y3=0:POKE PL
    X+3,X3:POKE PLY+3,Y3:T3=P3:P3=0:M=M-1
270 POKE 53278,1
300 IF M<=0 THEN GOTO 380
310 IF P2<>0 THEN GOTO 345
320 X=INT(RND(0)*3):IF X<>0 THEN GOTO 345
330 P2=T2:Y2=INT(RND(0)*180+32):X2=INT(RND(O
    )*130+48):POKE PLX+2,X2:POKE PLY+2,Y2
345 IF P3<>0 THEN GOTO 380
346 X=INT(RND(0)*3):IF X<>0 THEN GOTO 380
350 P3=T3:Y3=INT(RND(0)*180+32):X3=INT(RND(O
    )*130+48):POKE PLY+3,Y3:POKE PLX+3,X3
380 IF P2=0 THEN GOTO 415
390 IF X2>X0 AND P2=1 THEN P2=-1:D=USR(260,P
    MBASE+1537+Y2,PMBASE+57):GOTO 410
400 IF X2<X0 AND P2=-1 THEN P2=1:D=USR(260,P
    MBASE+1537+Y2,PMBASE+65)
410 X2=X2+4*(X2<X0)-4*(X2>X0):Y2=Y2+4*(Y2<Y0
    )-4*(Y2>Y0):POKE PLX+2,X2:POKE PLY+2,Y2
415 IF P3=0 THEN GOTO 60
420 IF X3>X0 AND P3=1 THEN P3=-1:D=USR(260,P
    MBASE+1793+Y3,PMBASE+57):GOTO 440
430 IF X3<X0 AND P3=-1 THEN P3=1:D=USR(260,P
    MBASE+1793+Y3,PMBASE+65)
440 X3=X3+4*(X3<X0)-4*(X3>X0):Y3=Y3+4*(Y3<Y0
    )-4*(Y3>Y0):POKE PLX+3,X3:POKE PLY+3,Y3
450 GOTO 60
500 D=USR(260,PMBASE+1024+Y0,PMBASE+33):NS=N
    S-1:SOUND 1,200,4,10:FOR W=1 TO 500:NEXT
     W:SOUND 1,0,0,0
510 IF NS<>0 THEN GOTO 540
520 NS=5:M=MON
530 IF STICK(0)=15 THEN GOTO 530
540 POKE DL4,136:POKE DL5,RT+1:POKE PLX,116:
    POKE PLY,95:SF=1:D=USR(260,PMBASE+1024+9
    5,PMBASE+1)
545 X0=116:Y0=95:NUML=136:NUMH=RT+1
```

```
550 X2=0:Y2=0:X3=0:Y3=0:POKE PLX+2,X2:POKE P
    LY+2,Y2:POKE PLX+3,X3:POKE PLY+3,Y3:BOT=
    0
555 IF P2<>0 THEN T2=P2:P2=0
556 IF P3<>0 THEN T3=P3:P3=0
560 MF=0:D=USR(260,PMBASE+1280+Y1,PMBASE+41)
    :POKE 53278,1:FOR W=1 TO 500:NEXT W:GOTO
     60
600 SOUND 1,40,10,8:FOR W=1 TO 30:NEXT W:SOU
    ND 1,32,10,8:FOR W=1 TO 30:NEXT W
610 SOUND 1,26,10,8:FOR W=1 TO 30:NEXT W:SOU
    ND 1,22,10,8:FOR W=1 TO 30:NEXT W
620 SOUND 1,32,10,8:FOR W=1 TO 30:NEXT W:SOU
    ND 1,26,10,8:FOR W=1 TO 70:NEXT W:SOUND
    1,0,0,0
630 IF BOT=1 THEN RETURN
640 GOTO 530
1000 REM INITIALIZE VBLANK PM
1010 FOR I=1536 TO 1706:READ A:POKE I,A:NEXT
      I

1020 FOR I=1774 TO 1787:POKE I,0:NEXT I
1030 PM=PEEK(106)-16:PMBASE=256*PM
1040 FOR I=PMBASE TO PMBASE+2047:POKE I,0:NE
     XT I
1050 FOR I=PMBASE+1025 TO PMBASE+1029:READ A
     :POKE I,A:NEXT I
1060 FOR I=PMBASE+1537 TO PMBASE+1545:READ A
     :POKE I,A:NEXT I:RESTORE 3010
1065 FOR I=PMBASE+1793 TO PMBASE+1800:READ A
     :POKE I,A:NEXT I
1070 POKE 704,18:POKE 705,226:POKE 706,179:P
     OKE 707,82
1075 FOR I=PMBASE+1 TO PMBASE+72:READ A:POKE
      I,A:NEXT I
1080 PLX=53248:PLY=1780:PLL=1784
1090 POKE 559,62:POKE 623,1:POKE 1788,PM+4:P
     OKE 53277,3:POKE 54279,PM
1100 X=USR(1696)
1110 POKE PLL,8:POKE PLL+1,8:POKE PLL+2,8:PO
     KE PLL+3,8
1115 X0=116:Y0=95:X2=190:Y2=175:X3=170:Y3=17
     5
1120 POKE PLX,X0:POKE PLY,Y0:POKE PLX+2,X2:P
     OKE PLY+2,Y2:POKE PLX+3,X3:POKE PLY+3,Y
     3
1130 FOR A=260 TO 284:READ I:POKE A,I:NEXT A
1140 RETURN
2000 REM DATA FOR VBLANK INTERRUPT
     {8 SPACES}ROUTINE-SEE COMPUTE! FEB. 198
     2
```

```
2010 DATA 162,3,189,244,6,240,89,56,221,240,
     6,240,83,141,254,6,106,141
2020 DATA 255,6,142,253,6,24,169,0,109,253,6
     ,24,109,252,6,133,204,133
2030 DATA 206,189,240,6,133,203,173,254,6,13
     3,205,189,248,6,170,232,46,255
2040 DATA 6,144,16,168,177,203,145,205,169,0
     ,145,203,136,202,208,244,76,87
2050 DATA 6,160,0,177,203,145,205,169,0,145,
     203,200,202,208,244,174,253,6
2060 DATA 173,254,6,157,240,6,189,236,6,240,
     48,133,203,24,138,141,253,6
2070 DATA 109,235,6,133,204,24,173,253,6,109
     ,252,6,133,206,189,240,6,133
2080 DATA 205,189,248,6,170,160,0,177,203,14
     5,205,200,202,208,248,174,253,6
2090 DATA 169,0,157,236,6,202,48,3,76,2,6,76
     ,98,228,0,0,104,169
2100 DATA 7,162,6,160,0,32,92,228,96
3000 DATA 128,64,127,24,48
3010 DATA 1,54,52,248,28,103,74,138
3020 DATA 128,64,127,24,48,0,0,0,1,2,254,24,
     12,0,0,0,224,56,224,0,0,0,0,7,28,7,0,
     0,0,0,0
3050 DATA 219,219,60,231,231,60,219,219
3060 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0
3070 DATA 1,54,52,248,28,103,74,138,128,108,
     44,31,56,230,82,82
4000 DATA 104,104,133,204,104,133,203,104,13
     3,207,104,133,206,160,0,177,206,145,203
     ,200,192,8,208,247,96
4900 RT=PEEK(106):RT=RT-16:POKE 106,RT
4910 POKE 559,0:FOR I=RT*256 TO (RT+16)*256:
     POKE I,0:NEXT I:GRAPHICS 3+16:COLOR 1:P
     OKE 765,1
4920 POKE 89,RT+2:POKE 88,0:GOSUB 5100
4930 POKE 89,RT+2:POKE 88,240:GOSUB 5100
4940 POKE 89,RT+3:POKE 88,224:GOSUB 5200
4950 POKE 89,RT+4:POKE 88,208:GOSUB 5300
4951 POKE 89,RT+5:POKE 88,192:GOSUB 5100
4952 POKE 89,RT+6:POKE 88,176:GOSUB 5200
4953 POKE 89,RT+7:POKE 88,160:GOSUB 5300
4954 POKE 89,RT+8:POKE 88,144:GOSUB 5100
4955 POKE 89,RT+9:POKE 88,128:GOSUB 5200
4956 POKE 89,RT+10:POKE 88,112:GOSUB 5300
4957 POKE 89,RT+11:POKE 88,96:GOSUB 5100
4958 POKE 89,RT+12:POKE 88,80:GOSUB 5200
4960 POKE 89,RT+13:POKE 88,64:GOSUB 5400
4965 POKE 89,RT+14:POKE 88,48:GOSUB 5500
4970 DL=PEEK(560)+256*PEEK(561):DL4=DL+4:DL5
```

```
     =DL+5
4980 POKE DL5,RT+1:POKE DL4,136:POKE 559,34
4990 RETURN
5100 PLOT 0,0:PLOT 29,0:DRAWTO 39,0:PLOT 0,1
     :PLOT 1,1:PLOT 30,1:DRAWTO 39,1:PLOT 0,
     2:DRAWTO 2,2:PLOT 31,2:DRAWTO 39,2:PLOT
      0,3:DRAWTO 3,3:PLOT 32,3
5105 DRAWTO 39,3
5110 PLOT 0,4:DRAWTO 6,4:PLOT 34,4:DRAWTO 39
     ,4:PLOT 0,5:DRAWTO 7,5:PLOT 32,5:DRAWTO
      39,5:PLOT 0,6:DRAWTO 8,6:PLOT 31,6:DRA
     WTO 39,6:PLOT 0,7:DRAWTO 7,7
5120 PLOT 32,7:DRAWTO 39,7:PLOT 0,8:DRAWTO 5
     ,8:PLOT 34,8:DRAWTO 39,8:PLOT 0,9:DRAWT
     O 4,9:PLOT 35,9:DRAWTO 39,9:PLOT 0,10:D
     RAWTO 5,10:PLOT 34,10:DRAWTO 39,10
5130 PLOT 0,11:DRAWTO 7,11:PLOT 32,11:DRAWTO
      39,11:PLOT 0,12:DRAWTO 8,12:PLOT 31,12
     :DRAWTO 39,12:PLOT 0,13:DRAWTO 7,13:PLO
     T 32,13:DRAWTO 39,13
5140 PLOT 0,14:DRAWTO 5,14:PLOT 34,14:DRAWTO
      39,14:PLOT 0,15:DRAWTO 4,15:PLOT 34,15
     :DRAWTO 39,15:PLOT 0,16:DRAWTO 5,16:PLO
     T 34,16:DRAWTO 39,16
5150 PLOT 0,17:DRAWTO 7,17:PLOT 32,17:DRAWTO
      39,17:PLOT 0,18:DRAWTO 8,18:PLOT 31,18
     :DRAWTO 39,18:PLOT 0,19:DRAWTO 7,19:PLO
     T 32,19:DRAWTO 39,19
5160 PLOT 0,20:DRAWTO 5,20:PLOT 34,20:DRAWTO
      39,20:PLOT 0,21:DRAWTO 4,21:PLOT 35,21
     :DRAWTO 39,21:PLOT 0,22:DRAWTO 5,22:PLO
     T 34,22:DRAWTO 39,22
5170 PLOT 0,23:DRAWTO 7,23:PLOT 32,23:DRAWTO
      39,23
5180 RETURN
5200 PLOT 12,23:DRAWTO 16,19:DRAWTO 16,8:DRA
     WTO 8,0:DRAWTO 0,0:POSITION 0,23:XIO 18
     ,#6,0,0,"S:":PLOT 39,19:DRAWTO 39,8:DRA
     WTO 23,8
5210 POSITION 23,19:XIO 18,#6,0,0,"S:"
5220 X=31:FOR Y=0 TO 7:PLOT X,Y:DRAWTO 39,Y:
     X=X-1:NEXT Y
5230 X=24:FOR Y=20 TO 23:PLOT X,Y:DRAWTO 39,
     Y:X=X+1:NEXT Y
5240 RETURN
5300 PLOT 17,23:DRAWTO 7,12:DRAWTO 18,12:DRA
     WTO 22,8:DRAWTO 20,8:DRAWTO 12,0:DRAWTO
      0,0:POSITION 0,23:XIO 18,#6,0,0,"S:"
5310 X=27:FOR Y=0 TO 7:PLOT X,Y:DRAWTO 39,Y:
     X=X+1:NEXT Y
```

```
5320 X=29:FOR Y=8 TO 13:PLOT X,Y:DRAWTO 39,Y
     :X=X-1:NEXT Y
5330 X=33:FOR Y=14 TO 23:PLOT X,Y:DRAWTO 39,
     Y:X=X-1:NEXT Y
5340 RETURN
5400 PLOT 11,23:DRAWTO 11,13:DRAWTO 3,13:DRA
     WTO 3,8:DRAWTO 16,0:DRAWTO 0,0:POSITION
      0,23:XIO 18,#6,0,0,"S:"
5410 X=24:FOR Y=0 TO 10:PLOT X,Y:DRAWTO 39,Y
     :X=X+1:NEXT Y
5420 PLOT 34,11:DRAWTO 39,11:PLOT 34,12:DRAW
     TO 39,12
5430 X=27:FOR Y=13 TO 23:PLOT X,Y:DRAWTO 39,
     Y:NEXT Y:PLOT 0,23:DRAWTO 39,23
5440 COLOR 3:PLOT 18,22:DRAWTO 20,22:PLOT 17
     ,22:DRAWTO 20,22
5450 RETURN
5500 COLOR 1:FOR Y=0 TO 23:PLOT 0,Y:DRAWTO 3
     9,Y:NEXT Y:RETURN
```

CHAPTER FOUR

# APPLICATIONS

# A Simple Text Editor

Osvaldo Ramirez

*You can use this line-oriented Editor for simple word processing, or even to edit BASIC programs stored with the LIST"D:" command.*

This program is a modification of Arnie Lee's and Steve Gradigan's text editor published in **COMPUTE!**, issues no. 9 and 20. The original program was written for the PET machine. This modification is for the Atari 800 and will work with an Epson MX 80 F/T printer with the Graftrax option. The program itself requires 9676 bytes, but goes up to 31865 bytes once the buffer is initiated. The buffer accommodates five pages of 54 lines each, 80 characters per line. Two extra lines were added for string manipulations. Each page requires 4405 bytes. This amount may be reduced for machines with less memory by changing the DIM statement in line 20 and the string clearing routine in line 30. The translation was accomplished using Charles Brannon's suggestions in **COMPUTE!**, issue no. 16 and Teri Li's in *Byte* of January 1981.

I added a few extra routines that enhance an already useful program. Each function now has its own screen. In addition to the bell warning that there are five spaces left to fill the input string, there are now five reverse video spaces to be overwritten so that the user has visual control of the spaces left for proper hyphenation, when needed. If the string length (the M value) exceeds 76, a new line will be displayed on the TV. It will not affect the buffer or the printing.

A new line exchange function and a block move of lines were added. These two new routines increase editing flexibility. They can be accessed with the B and the E commands, respectively.

The loading function will automatically append to whatever program is resident in the machine's memory if it is not deleted before loading the new text.

The printing routine formats to 54 lines per page and exits printing with a top of form command. It will number pages automatically at the top, with right justification starting at the second page while retaining the 54 line format. It stops at the end of each page and prompts the user to continue when ready. This feature allows for the insertion of a new page if you are using cut paper instead of fanfold. The printer will respond to the Atari control characters, allowing for different fonts in the same line.

The string change function will accept from one line up to the entire document for search and substitution. It will display on the screen the line selected or the first line of the block of lines to search. The program takes about two minutes to scan each page.

The program was renumbered to maintain the same routine locations as in the original listing of Lee and Gradijan. This way you can refer to their original articles for documentation. The renumber was done using one of the utility programs from the Atari Program Exchange.

When keying in this program, remember the five reverse video characters in line 10100, the five left arrow characters in line 10080, and one down arrow in line 21010 right after the colon.

## PROGRAM. A Simple Text Editor For The Atari

```
10 REM -A SIMPLE TEXT EDITOR FOR ATARI
11 REM BY O.RAMIREZ-MARCH 1982
12 REM ORIGINAL PROGRAMS BY A.LEE AND S.GRAD
   IJAN
13 REM COMPUTE #9 AND #20
20 DIM T$(21760),L$(80),B$(80),A$(13)
30 T$(1)=" ":T$(21760)=" ":T$(2)=T$
40 DIM DM$(2),FR$(80),TS$(80)
50 L$(1)=" ":L$(80)=" ":L$(2)=L$
60 DIM FI$(12),NL$(80),F$(2)
70 OPEN #1,4,0,"K:"
80 POKE 82,0
90 LL=1:M=79
100 A$="ABCDEFILMPQRS"
110 ? "{CLEAR}{10 SPACES}SOFTWARE LINE EDITOR
    "
120 ? :? :? "FUNCTIONS:"
130 ?
140 POKE 85,10:? "A.append to end of text/st
    art"
150 POKE 85,15:? "@ =5 space tab"
160 POKE 85,15:? "@+RETURN = skip a line"
170 POKE 85,10:? "B.block move"
180 POKE 85,10:? "C.change string"
190 POKE 85,10:? "D.delete a line"
200 POKE 85,10:? "E.exchange lines"
210 POKE 85,10:? "F.filer commands"
220 POKE 85,10:? "I.insert before line"
230 POKE 85,10:? "L.list line(s)"
240 POKE 85,10:? "M.menu display"
250 POKE 85,10:? "P.print lines"
260 POKE 85,10:? "Q.quit editor"
270 POKE 85,10:? "R.replace a line"
280 POKE 85,10:? "S.set margins"
290 ? :?
300 B=0:? "ENTER SELECTION-->";
310 GOTO 330
320 TRAP 40000:? :? "ENTER A,B,C,D,E,F,I,L,P
    ,Q,R,S,M -->";
330 GET #1,B:B$=CHR$(B):IF B$="" THEN 330
340 J=0:FOR I=1 TO 13:IF A$(I,I)=B$(1,1) THE
    N J=I
350 NEXT I:I=13
360 ? B$
370 IF J=0 THEN 320
380 ON J GOTO 1000,12000,2000,3000,11000,400
    0,5000,6000,110,9000,8000,7000,21000

1000 ? "{CLEAR}APPEND TO END OF TEXT OR STAR
```

```
     T"
1010 ? LL;">"
1020 GOSUB 10000
1030 IF LEN(L$)=0 THEN 320
1040 T$((LL*80)-79,(LL*80))=L$
1050 LL=LL+1
1060 GOTO 1010
2000 ? "{CLEAR}CHANGE ";:GOSUB 16000
2010 IF HI=0 THEN 320
2020 J2=LO
2030 ? "CHANGE STRINGS->":? J2;">";T$(J2*80-
     79,J2*80-80+M):GOSUB 10000
2040 L=LEN(L$)
2050 IF L=0 THEN 320
2060 IF L<4 THEN 2000
2070 DM$=L$(1,1)
2080 IF L$(LEN(L$))<>DM$ THEN ? "*WRONG,TRY
     AGAIN":FOR I=1 TO 200:NEXT I:GOTO 2000
2090 J=0:FOR I=2 TO L-1
2100 IF L$(I,I)=DM$ THEN J=I
2110 NEXT I
2120 IF J=0 THEN 2000
2130 IF J=2 THEN 2000
2140 FR$=L$(2,J-1)
2150 IF J+1=L THEN TS$="":GOTO 2170
2160 TS$=L$(J+1,L-1)
2170 F=LEN(FR$)
2180 POP :FOR I=LO TO HI
2190 T=M:S=1:NL$=""
2200 FOR J=1 TO T-F+1
2210 IF T$(I*80-80+J,I*80-80+J+F-1)<>FR$ THE
     N 2250
2220 IF J=1 THEN NL$=TS$:GOTO 2240
2230 NL$(LEN(NL$)+1)=T$(I*80-80+S,I*80-80+J-
     1):NL$(LEN(NL$)+1)=TS$
2240 S=J+F
2250 NEXT J
2260 IF S<>1 THEN NL$(LEN(NL$)+1)=T$(I*80-80
     +S,I*80-80+M+1):T$(I*80-79,I*80)=T$((LL
     +1)*80-79,(LL+1)*80)
2265 T$(I*80-79,I*80)=NL$
2270 NEXT I
2280 GOTO 320
3000 ? "{CLEAR}DELETE ";:GOSUB 16000
3010 IF DF=0 THEN 3060
3020 ? "DELETE THE ENTIRE FILE?(Y/N)";
3030 GET #1,B:B$=CHR$(B):IF B$=" " THEN 3030
3040 ? B$:IF B$="N" THEN 320
3050 IF B$<>"Y" THEN 3020
3060 IF HI>LL-1 THEN 320
```

```
3070 IF HI=LL-1 THEN TRAP 320:T$(HI*80-79,HI
     *80)=T$((LL+1)*80-79,(LL+1)*80):LL=LO:G
     OTO 320
3080 J=HI-LO+1
3090 FOR I=LO TO LL+1
3100 T$(I*80-79,I*80)=T$((I+J)*80-79,(I+J)*8
     0)
3110 NEXT I
3120 LL=LL-(HI-LO)-1
3130 GOTO 320
4000 ? "{CLEAR}FILE COMMANDS ENTER L=LOAD,S-
     SAVE->";
4010 GET #1,B:B$=CHR$(B):IF B$="" THEN 4010
4020 IF B$=CHR$(155) THEN ? "":GOTO 320
4030 IF B$<>"L" AND B$<>"S" THEN ? :GOTO 400
     0
4040 ? B$:F$=B$
4045 ? :? :POKE 85,10:? "INSERT STORAGE DISK
     ":? :?
4050 POKE 85,20:? "|--------.---|"
4060 POKE 85,5:? "ENTER FILENAME->";
4070 GOSUB 10000
4080 IF LEN(L$)=0 THEN 320
4090 IF LEN(L$)>12 THEN ? "USE LESS THAN 12
     CHARACTERS":GOTO 4050
4100 FI$=L$
4110 B$="D:"
4120 B$(LEN(B$)+1)=FI$
4130 IF F$="L" THEN 4210
4140 IF LL=1 THEN ? "NO FILE TO SAVE":GOTO 3
     20
4150 CLOSE #3:OPEN #3,8,0,B$
4160 FOR I=1 TO LL-1
4170 ? #3;T$(I*80-79,I*80)
4180 NEXT I
4190 CLOSE #3:? :? FI$;" SAVED"
4200 GOTO 320
4210 CLOSE #3:TRAP 4280:OPEN #3,4,0,B$
4220 IF LL>1 THEN LL=LL-1:GOTO 4230
4225 LL=0
4230 LL=LL+1:T$(LL*80-79,LL*80)=" "
4240 L$=" "
4250 TRAP 4270:INPUT #3;L$:T$(LL*80-79,LL*80
     )=L$
4260 GOTO 4230
4270 TRAP 40000:CLOSE #3:GOTO 320
4280 CLOSE #3:TRAP 40000:? "FILE NOT ON DISK
     ":GOTO 320
5000 ? "{CLEAR}INSERT BEFORE ";:GOSUB 17000
5010 IF LO>LL OR LO<1 THEN 5000
```

```
5020 ? :? LO;">";
5030 GOSUB 10000
5040 IF LEN(L$)=0 THEN 320
5050 LL=LL+1
5060 FOR I=LL TO LO STEP -1
5070 IF I=1 THEN 5090
5080 T$(I*80-79,I*80)=T$((I-1)*80-79,(I-1)*8
     0)
5090 NEXT I
5100 T$(LO*80-79,LO*80)=T$((LL+2)*80-79,(LL+
     2)*80)
5110 T$(LO*80-79,LO*80-80+M)=L$
5120 LO=LO+1
5130 GOTO 5020
6000 ? "{CLEAR}LIST ";:GOSUB 16000
6010 IF HI=0 THEN 320
6020 FOR J=LO TO HI
6030 ? J;">";T$(J*80-79,J*80-80+M)
6040 IF J/10=INT(J/10) THEN POSITION 3,23:?
     "HIT E TO END,RETURN TO CONTINUE->";:GE
     T #1,B:? CHR$(B):IF B=69 THEN 320
6050 NEXT J
6060 GOTO 320
7000 ? "{CLEAR}REPLACE ";:GOSUB 17000
7010 IF LO>=LL OR LO<1 THEN 7000
7020 ? "LINE TO REPLACE:"
7030 ? LO;">";T$(LO*80-79,LO*80)
7040 ? :? LO;">";
7050 GOSUB 10000
7060 IF LEN(L$)=0 THEN 320
7080 T$(LO*80-79,LO*80)=T$((LL+1)*80-79,(LL+
     1)*80)
7090 T$(LO*80-79,LO*80)=L$
7100 GOTO 320
8000 ? "{CLEAR}":POSITION 4,10:? "LEAVE EDIT
     OR:ARE YOU SURE? (Y/N)";
8010 GET #1,B:B$=CHR$(B):IF B$="" THEN 8010
8020 IF B$<>"Y" AND B$<>"N" THEN 8000
8030 IF B$="N" THEN ? :? :? :? :? :GOTO 320
8040 POKE 752,1:POSITION 9,12:? CHR$(253);"E
     ND OF EDITOR PROGRAM...":FOR I=1 TO 200
     :NEXT I:GRAPHICS 0:END
9000 CLOSE #2:TRAP 9200:OPEN #2,8,0,"P:"
9010 ? "{CLEAR}PRINT ";:GOSUB 16000
9020 ? "NUMBER OF SPACES BETWEEN LINES(1-2)"
     ;:INPUT S1

9030 ? "IF THIS IS THE FIRST PAGE ENTER 1"
9040 ? "OTHERWISE ENTER APPROPIATE NO....";:
     INPUT PP
9050 IF HI=0 THEN 320
```

```
9060 CT=0
9070 ? #2;CHR$(27);"D";CHR$(SP);CHR$(0)
9080 FOR I=LO TO HI
9090 ? #2;CHR$(137);T$(I*80-79,I*80-80+M)
9100 CT=CT+1
9110 IF S1=2 THEN ? #2:CT=CT+1
9120 IF CT/54=INT(CT/54) THEN GOSUB 23000
9130 NEXT I
9140 ? #2;CHR$(12)
9150 CLOSE #2:GOTO 320
9200 ? :? :POKE 85,10:? "TURN PRINTER ON":TR
     AP 40000:GOTO 320
10000 L$=""
10010 REM
10020 GET #1,B:B$=CHR$(B):IF B$="" THEN 1002
      0
10030 IF B$=CHR$(155) THEN POKE 752,0:? :RET
      URN
10040 IF B$=CHR$(126) THEN L$(LEN(L$))="":?
      B$;:GOTO 10110
10050 IF B$="@" THEN B$="{5 SPACES}"
10060 IF LEN(L$)=M-6 THEN ? CHR$(253);
10070 IF LEN(L$)=M THEN POKE 752,0:GOTO 1012
      0
10080 IF LEN(L$)=M-5 THEN ? "{5 LEFT}";
10090 L$(LEN(L$)+1)=B$:? B$;
10100 IF LEN(L$)=M-5 THEN POKE 752,1:? "
      {5 SPACES}";
10110 GOTO 10010
10120 ? "ERROR-LINE TUNCATED":RETURN
11000 ? "EXCHANGE LINES"
11010 ? "ENTER LOWER NUMBER FIRST"
11030 ? "ENTER ";:GOSUB 16000
11040 IF L=0 THEN 320
11060 TS$=" ":TS$=T$(LO*80-79,LO*80)
11070 T$(LO*80-79,LO*80)=T$(HI*80-79,HI*80)
11080 T$(HI*80-79,HI*80)=" "
11090 T$(HI*80-79,HI*80)=TS$:TS$=" "
11100 ? LO;">";T$(LO*80-79,LO*80)
11110 ? :? HI;">";T$(HI*80-79,HI*80)
11120 GOTO 320
12000 ? "{CLEAR}MOVE A BLOCK OF LINES"
12010 ? "INSERT BEFORE LINE NO.-> ";:INPUT B
      $:IF LEN(B$)=0 THEN 320
12015 A=VAL(B$):B$=""
12020 ? :? "BLOCK TO MOVE ";:GOSUB 16000
12030 IF LO=A THEN 12000
12040 CTR=0
12050 IF LO<A THEN 12200
12060 IF HI>LL-1 THEN 320
```

```
12080 I=0:FOR I=LL+1 TO A STEP -1
12090 T$(I*80-79,I*80)=T$((I-1)*80-79,(I-1)*
      80)
12100 NEXT I
12110 T$(A*80-79,A*80)=T$((HI+1)*80-79,(HI+1
      )*80)
12120 I=0:FOR I=HI+1 TO LL+1
12130 T$(I*80-79,I*80)=T$((I+1)*80-79,(I+1)*
      80)
12140 NEXT I
12160 CTR=CTR+1
12170 IF CTR=HI-LO+1 THEN 320
12180 GOTO 12080
12200 I=0:FOR I=LL+1 TO A STEP -1
12210 T$(I*80-79,I*80)=T$((I-1)*80-79,(I-1)*
      80)
12220 NEXT I
12230 T$(A*80-79,A*80)=T$(LO*80-79,LO*80)
12240 I=0:FOR I=LO TO LL+1
12250 T$(I*80-79,I*80)=T$((I+1)*80-79,(I+1)*
      80)
12260 NEXT I
12280 CTR=CTR+1
12290 IF CTR=HI-LO+1 THEN 320
12300 GOTO 12200
16000 ? "RANGE(LOW-HIGH)=> ";
16010 GOSUB 10000
16020 LO=1:HI=LL-1
16030 L=LEN(L$)
16040 DF=0:IF L=0 THEN DF=-1:GOTO 16170
16050 J=0:FOR I=1 TO L
16060 B$=L$(I,I)
16070 IF B$>="0" AND B$<="9" THEN 16110
16080 IF B$="-" THEN J=I:GOTO 16110
16100 J=99:I=99
16110 NEXT I
16120 IF J=99 THEN 16000
16130 IF J=0 THEN LO=VAL(L$):HI=LO:RETURN
16140 IF J>1 THEN LO=VAL(L$(1,J-1))
16150 IF J<L THEN HI=VAL(L$(J+1,LEN(L$)))
16160 IF LO>HI THEN 16000
16170 RETURN
17000 ? "-LINE NO.->";
17010 GOSUB 10000
17020 L=LEN(L$)
17030 IF L=0 THEN 320
17040 J=0
17050 FOR I=1 TO L
17060 B$=L$(I,I)
17070 IF B$>="0" AND B$<="9" THEN 17090
```

```
17080 J=99:I=L
17090 NEXT I
17100 IF J=99 THEN 17000
17110 LO=VAL(L$)
17120 RETURN
21000 ? "{CLEAR}SET MARGINS"
21010 POKE 85,7:? "MARGIN SIZE:{DOWN}N.none"
21020 POKE 85,19:? "S.small (0.5 in.)"
21030 POKE 85,19:? "M.medium (1.0 in.)"
21040 POKE 85,19:? "L.large (1.5 in.)"
21050 POKE 85,19:? "O.own design"
21060 B=0:? :? "SELECT MARGIN SIZE->";
21070 GET #1,B:B$=CHR$(B):IF B$="" THEN 2107
      0
21080 ? B$
21085 IF B$<>"N" AND B$<>"S" AND B$<>"M" AND
       B$<>"L" AND B$<>"O" THEN 21060
21090 IF B$="N" THEN M=79:SP=0:GOTO 320
21100 IF B$="S" THEN M=74:SP=3:GOTO 320
21110 IF B$="M" THEN M=64:SP=8:GOTO 320
21120 IF B$="L" THEN M=54:SP=13:GOTO 320
21130 IF B$="O" THEN ? :? "INCHES FOR LEFT M
      ARGIN=";:INPUT SP
21140 ? :? "INCHES FOR RIGHT MARGIN=";:INPUT
       RM
21150 SP=INT((SP-0.2)*10):M=79-SP-INT((RM-0.
      25)*10)
21160 GOTO 320
23000 ? :? "PRESS RETURN TO PRINT NEXT PAGE"
      ;:INPUT B$
23010 ? #2;CHR$(12)
23020 P1=6:IF PP>9 THEN P1=7
23030 PP=PP+1:CT=CT+1
23040 FOR I1=1 TO SP+M-P1:PUT #2,32:NEXT I1:
      ? #2;"Page ";PP
23045 IF S1=2 THEN ? #2:CT=CT+1
23050 RETURN
```

# The Atari Keyboard Speaks Out

Walter M. Lee

*How to use the console speaker – Atari's* fifth *voice. Here's an explanation of the Atari built-in loudspeaker and a user-controllable BASIC program to play music without using the TV speaker.*

One of the frequently unused features of the Atari computer is the keyboard loudspeaker. Many of the sound effects created on the Apple II+ keyboard loudspeaker can also be generated on the Atari. The four-voice audio output of the Atari to the television is more flexible than the keyboard speaker. However, if your display is a monitor instead of a television set, it may not support the four-voice audio output. The keyboard loudspeaker is then a practical means for audio output or feedback. In addition, the keyboard loudspeaker allows the television or monitor to be turned off while the Atari is executing a long program. The loudspeaker can then signal when the job is finished.

The Atari keyboard loudspeaker is accessible from the BASIC cartridge. A simple PRINT statement will create a buzzer.

```
10 PRINT "{BELL}";
```

This buzzer has a fixed tone and duration. To create a variable duration, one must POKE zero into the speaker register, CONSOL, at location 53279 decimal. The duration is set by the number of times CONSOL is set to zero.

```
10 INPUT N
20 FOR I=0 TO N:POKE 53279,0:NEXT I
30 GOTO 10
```

Creating pure pitches on the Atari keyboard loudspeaker requires more work. A loudspeaker produces sound by creating waves of alternating high and low pressure in the air. The loudspeaker does this by moving a diaphragm (the loudspeaker's cone) forward and backward. As the diaphragm moves forward it squeezes the air in front of it, causing a region of high pressure. As the diaphragm moves backward, the air rushes in to fill the space left behind the moving diaphragm and creates a region of low pressure. These pressure waves radiate out from the loudspeaker in the direction of propagation; i.e., sound waves are longitudinal waves. Storing 8 into CONSOL pushes the diaphragm one way, and storing 0 into CONSOL pushes the diaphragm the other way.

The time delay between switching CONSOL from 8 to 0 constitutes the period of the sound wave; therefore, it designates the frequency generated by the sound wave. The shorter the time delay, the higher the frequency. The Vertical Blank Interrupt stores an 8 into CONSOL every 1/60 of a second. This is why Program 1 works. However, this also prevents the keyboard loudspeaker from generating any pitch other than 60 hertz (hertz = cycles/second). Fortunately, we do not have to disable the entire Vertical Blank Interrupt to create variable pitches on the loudspeaker. The Vertical Blank Interrupt is broken into two stages. During critical code sections (e.g., I/O routines), the Atari Operating System will defer the second stage of the Vertical Blank Interrupt. This is done by setting the CRITIC flag, at location 66 decimal, to a nonzero value. The second stage stores 8 into CONSOL. The first stage updates the real-time clock, the ATTRACT mode, and the system countdown timer 1. The shadow registers, the game controllers, and the system countdown timers two through five are disabled with the second stage. To regain the second stage, we set CRITIC = 0, which is its normal state.

The time delay for the Atari has an intrinsic pitch distortion. This is due to the Atari Display processor, ANTIC, which "steals" machine cycles from the 6502 in order to generate the display on the television and to refresh memory. This is called Direct Memory Access (DMA). The DMA cannot be accounted for in the delay loop and causes the pitch to get out of synchronization. The accuracy of the pitch generated must be sacrificed if we are to maintain a display. Graphics mode 3 through 6 (BASIC) seem to have the least effect on pitch distortion. To create the purest tone possible, the screen must be turned off. This is achieved by storing zero into DMACTL, at location 54272 decimal, after disabling the shadow registers by setting CRITIC = 1. Program 1 creates various low tones on the keyboard loudspeaker.

In machine language, the frequency range is extended. The upper frequencies are increased by a much greater extent than the lower frequencies. The lower frequencies degrade into "clicks," which is normal for square wave sound synthesis.

Most users will want to use this technique in BASIC. I have written an Atari BASIC USR function to control the keyboard loudspeaker. Program 2 puts this USR function in memory locations 1536 to 1600 decimal. The USR function

DUMMY = USR(1536,I,J)

sets I = pitch and J = duration. The pitch can be from 1 (the highest) to 255 (the lowest). Setting I = 0 is equivalent to setting I = 256. The variable *J* has a range from 0 to 65,535. The longest duration is when J = 65,535. The pitch distortion should be around 10% when the screen is off. Program 2 does a frequency range test and then plays a perceptible version of the ABC song. A table of pitch values for musical notes is also included in the program. To create other tunes, I have included a table of corresponding musical notes and their approximate pitch numbers. These values were calculated from a *CRC Handbook of Chemistry and Physics* and may not be correct. My checking was hampered by the fact that I do not have a musical ear. The Pitch number is the reciprocal of the product

$$\text{pitch} = \frac{1}{\text{musical note frequency} * \text{cycle speed} * 5 \text{ cycles}}$$

The cycle speed is .8517 microseconds per machine cycle, which is much slower than the official speed of .56 microseconds per machine cycle. The Atari's 6502B main processor has an effective speed of 1.17Mhz in Graphics Mode 0 (BASIC). I calculated this speed by dividing the execution time of A = USR(1536,255,8191) by the number of cycles the 6502 must execute.

These programs present the fifth audio voice to Atari users. For users without a television or monitor audio output, these programs give the user a limited audio output without additional hardware. For the less ambitious programmer, lines 29100 to 30002 in Program 2 and the pitch table to musical notes are all you need.

## PROGRAM 1. The Atari Keyboard Speaks Out

```
100 C=53279
101 S1=8:S2=0:A=24:B=2:D=48
102 CRITIC=66:DMACTL=54272
200 POKE CRITIC,66:POKE DMACTL,0
222 FOR J=0 TO A STEP B:FOR K=0 TO D:POKE C,
    S1:FOR I=0 TO J:NEXT I:POKE C,S2:NEXT K:
    NEXT J
240 POKE CRITIC,0
```

**PROGRAM 2. The Atari Keyboard Speaks Out**

```
90 DIM PITCH(32,2)
99 GRAPHICS 2+16:? #6;"FREQUENCY RANGE TEST
   "
100 GOSUB 29100:REM MAKE SUBROUTINE
101 RESTORE 110
102 REM PITCH#=1/(FREQ*.85^-(7*5))
103 REM EQUAL TEMPERED CHROMATIC
104 REM SCALE:A(4)=440hz
105 REM FROM A#(5)=932hz to
106 REM  C(8)=4186hz.see CRC Handbook
107 REM of Chem.and Physics 58th ed.
108 REM page E-48
110 DATA 252,238,224,212,200,189,178,168,159
    ,150,141,133,126,119,112,106,100,94,89,8
    4,79,75,71,67,63,59,56,53
200 FOR I=1 TO 28:READ A
210 PITCH(I,1)=A:PITCH(I,2)=1024+(I-1)*127:N
    EXT I
290 REM FREQUENCY RANGE TEST
300 FOR I=1 TO 28:A=USR(1536,PITCH(I,1),PITC
    H(I,2)):NEXT I
390 REM SONG  A. B. C.
400 RESTORE 410
401 POSITION 0,3:? #6;"ABC"
410 DATA 3,3,10,10,12,12,10,8,8,7,7,5,5,5,5,
    3,10,10,8,8,7,7,5,10,10,8,8,7,7,5,3,3,10
    ,10,12,12,10,8,8,7,7,5,5,3
420 FOR I=1 TO 44:READ A:A=USR(1536,PITCH(A,
    1),PITCH(A,2)):NEXT I
29099 END
29100 RESTORE 30000
29101 FOR I=1536 TO 1600:READ A:POKE I,A:NEX
      T I:POKE I+1,0:RETURN
30000 DATA 133,203,104,104,133,205,104,133,2
      04,104,133,207,104,133,206,165,203,72,
      138,72,152,72
30001 DATA 169,1,133,66,166,206,169,8,141,31
      ,208,164,204,136,208,253,140,31,208,20
      2,208,240
30002 DATA 169,0,197,207,240,5,198,207,76,26
      ,6,104,168,104,170,104,169,0,133,66,96
```

# Atari Screen As Strip Chart Recorder

Helmut Schmidt

*This program lets you simulate a two-pen strip chart recorder. There is information here on coarse scrolling, vertical blank interrupts, and even ESP!*

I will describe a subroutine that gives a display like a strip chart recorder where two colored pens write on a moving roll of paper. We will make the screen scroll up, while at the bottom the coordinates X1 and X2 of the two pens are entered continuously. This display can be very useful to plot computer generated functions or to graph data provided from outside.

Using a 4K memory block as display area, the Atari allows us very easily to scroll this memory block as a closed loop past the screen display window. Remember that two bytes in the display list specify the current start address of the display memory. Using Graphics Mode 7+16 where each line holds 40 bytes, we get vertical scrolling if we regularly increment the start address in steps of 40. Beginning at the start address, the subsequent memory is displayed on the screen. But there is a hitch that comes in very handy. When the display memory reaches a 4K boundary, the display does not enter the next 4K block, but starts at the beginning of the initial 4K block. Thus, the memory block appears "wrapped around" in a continuous loop, provided that the start address stays within this block.

To display, for example, the lowest 4K memory block in this closed loop scroll fashion we can use the simple BASIC program:

```
10 GRAPHICS 7+16
20 LOC=PEEK(560)+256*PEEK(561)+4
30 L=0:H=0
40 L=L+40
50 IF L>255 THEN L=L-256:H=H+1
60 IF H>15 THEN H=H-16
70 POKE LOC,L:POKE LOC+1,H
80 GOTO 40
```

Unfortunately, this program needs two POKE steps to increment the display start address, so that an eye straining flicker may occur when a display interrupt falls between the two steps. This problem disappears if we advance the display window by a machine language program that is inserted in the Vertical Interrupt section at the end of each display cycle. Chapter 8 of *De Re Atari* tells specifically where to do this insertion. For reasons of economy and speed it is good to include into the machine language program the other housekeeping functions of the display – like entering the new coordinates at the bottom, and clearing the outdated entries (leaving the screen at the top).

Our machine program gets assembled (Program 1) into the second half of page six ($680-$6FF). The reader with an assembler cartridge can easily disassemble the program and study it in detail. For most practical purposes it is sufficient to know the location of a few parameters, so that by POKEing in new values we can set the pen coordinates, alter the scrolling speed, and stop and start the scrolling.

Here is a list of the useful POKE operations:

| | |
|---|---|
| POKE 1664,A | Sets the writing pens to positions |
| POKE 1665,B | A and B (Range 0 to 179) |
| POKE 1677,144 | Stops scrolling |
| POKE 1677,208 | Scrolling again |
| POKE 1666,80 | Extra fast scrolling |
| POKE 1666,40 | Back to normal scroll |
| POKE 1665,Z | Normal speed for Z=0. Slower for Z=1,3,7,15,31,63,127,255 |

Program 1 gives the three basic subroutines to handle the strip chart display.

1. GOSUB 800 has to be run first to enter the machine program into the lower half of page six.

2. When we actually want the scrolling display we call GOSUB 700. This tells the machine program where the display memory is (depending on RAM size), and then switches the machine program into the Vertical Interrupt cycle.

3. GOSUB 780 provides an orderly exit from the scrolling mode into Graphics 0.

To demonstrate the use of these subroutines in a most simple example, merge Program 2 with Program 1. The resulting program shows the traces of the two pens that are moved by Paddle 0 and 1. (If the paddles are not plugged in, or are set out of range, no trace is displayed.) To change parameters press RESET, POKE in the new values, and GOTO 100. Program 2 gives some more explanations.

## Strip Chart Application: Man-Computer Interaction

The merger of Program 3 with Program 1 gives our next program. Again we have two pens writing on the scrolling screen. One pen is stationary to mark the center line, and the other pen performs a sine-wave motion with randomly varying amplitude. Whenever the pen crosses the center line, a random decision is made on whether to increase or decrease the amplitude for the next half cycle (with upper and lower bounds for the amplitude).

After a certain number of cycles the swinging stops and a score is displayed. A positive or negative value indicates that the average amplitude of the swinging pen in this run was higher or lower than chance expectancy. Thus, by the laws of chance, positive and negative scores are equally likely.

Each of the binary decisions (increase or decrease) results from a combination of human decision and computer decision. The Atari provides in Location 53770 a rapidly changing sequence of quasi random numbers in the range from 0 to 255. At each crossing of the center line, the computer decides to increase or decrease the pen amplitude depending on whether or not the current random number is larger than 127. This determines the course of events as long as the human operator does not press the OPTION button on the console. If, however, the OPTION button is held down at the time of the crossing, then the computer's internal decision for increase or decrease is inverted.

To explore the program further, let us first run it without using the OPTION button. What determines the history of the pen oscillations is the internal state of the computer's quasi random number generator at the time when the START button is pushed. And because this internal state changes at a very rapid rate (in the megacycle range), the timing of the START button push has a decisive effect on the outcome. Pressing the START button only one-tenth of a second later would lead to a completely different history of the pen oscillations.

It is interesting to know that a "favorable" timing of the START button push can produce high scores. But we cannot use this knowledge for actually obtaining high scores. This is because we have no way of knowing the internal state of the computer's random generator. And even if we knew, we could not do the fast mental arithmetic to calculate the resulting pen movements. And even if we could, our fingers would not be fast enough to press the start button at precisely the right time.

Having the OPTION button available doesn't help either. Now we can change the history of a run while it is in progress. But since we do not know the computer's internal choice for the next amplitude, we cannot tell whether we should or should not invert the computer's decision by pushing the OPTION button. Thus, the arrangement provides a "fair game of chance" with no room for a systematic winning strategy.

Now we come to our main question: suppose you concentrate intensely on the swinging pen, trying to make it swing widely or to keep it still at the center, and you push the OPTION button whenever you feel it is right. Could you then by some "psychic" mechanism succeed more often than not? Here success means a positive or negative score when the aim was a high or a low swing amplitude, respectively.

The most recent report on the existence of such psychic effects can be found in the Proceedings of the IEEE (Vol. 70, No. 2, Feb. 1982, pp. 136-170). Its author, Robert Jahn, is the Dean of the Princeton Engineering Department. Thus, you don't have to feel too foolish in trying such an experiment.

The aim of a high or low swing amplitude is a rather plain and simple one. This may be important so that you can pay undivided one-pointed attention to the task and can get some feeling for the best mental approach. Keep the test sessions short and don't expect miracles. If, after some time, you could reach an average success rate of 65% or more of the runs in the desired direction, then your skill would be in high demand at several research laboratories.

**PROGRAM 1. Atari Screen As Strip Chart Recorder.**

```
697 REM
698 REM ***** GOSUB 700 STARTS SCROLLING ***
    **
699 REM
700 GRAPHICS 7+16:POKE 559,0:REM .....TV OFF
709 REM ...SET COLOR FOR BACKGROUND,TRACE X1
    ,TRACE X2
710 SETCOLOR 4,0,0:SETCOLOR 2,1,8:SETCOLOR 0
    ,8,4
720 PG=PEEK(106):REM .................NUMBER
     OF RAM PAGES
727 REM ........SCREEN DISPLAY STARTS AT 256
    *PG-4000
728 REM ........START WRITE 95 LINES(3800 WD
    S) FURTHER,I.E.
729 REM ........WRITESTART = 256*(PG-1)+56
730 POKE 203,56:POKE 204,PG-1:REM ....SET WR
    ITESTART
740 POKE 205,PEEK(560):POKE 206,PEEK(561)
750 KEEPL=PEEK(546):KEEPH=PEEK(547):REM ...S
    AVE PARAMETERS
760 POKE 546,135:POKE 547,6:REM ......MACHIN
    E PROGRAM START
770 POKE 559,34:RETURN :REM ..........TV ON
    AGAIN
771 REM
772 REM ***** GOSUB 780 EXITS FROM SCROLLING
     *****
773 REM
780 POKE 559,0:GRAPHICS 7+16:POKE 546,KEEPL:
    POKE 547,KEEPH
790 GRAPHICS 0:RETURN
797 REM
798 REM ***** GOSUB 800 LOADS MACHINE PROGRA
    M *****
799 REM
800 X1=1664:X2=1665:REM .............PEN CO
    ORDINATES
810 DATA 200,200,40,192,48,12,3,216,165,20,4
    1,0,24,208,66,165
811 DATA 203,109,130,6,133,203,144,26,165,20
    4,105,0,41,15,72,13
812 DATA 175,6,133,204,104,208,11,174,130,6,
    169,0,202,157,0,112
813 DATA 208,250,172,130,6,169,0,136,145,203
    ,208,251,160,4,177,205
814 DATA 109,130,6,145,205,200,177,205,105,0
    ,41,15,109,175,6,145
```

```
815 DATA 205,169,255,141,250,6,173,128,6,32,
    234,6,169,85,141,250
816 DATA 6,173,129,6,32,234,6,76,95,228,201,
    180,176,17,72,74
817 DATA 74,168,104,41,3,170,189,131,6,41,25
    5,17,203,145,203,96
820 POKE 559,0
830 FOR N=1664 TO 1791
840 READ DAT:POKE N,DAT:NEXT N
850 POKE 1711,PEEK(106)-16
860 POKE 559,34:RETURN
```

**PROGRAM 2. Atari Screen As Strip Chart Recorder.**

```
0 REM ........***** MERGE WITH LISTING 1 ***
  **
1 REM .......AT "RUN" WAITING PERIOD TO ENTE
  R PROGRAM,
2 REM .......THEN DISPLAY OF 2 PENS FROM PAD
  DLE INPUT.
3 REM .......NO TRACE IF INPUT OUT OF RANGE
  (0...179).
4 REM .......PUSH "SELECT"/"START" TO STOP/M
  OVE DISPLAY.
5 REM .......TO CHANGE PARAMETERS PRESS "SYS
  TEM RESET".
6 REM .......POKE 1675,Z WITH Z=0,1,3,7,15,3
  1,63,127,255.
7 REM .......Z=0 GIVES FASTEST(NORMAL) SCROL
  L SPEED.
8 REM .......FOR DOUBLE SPEED (SKIPPING ODD
  LINES) POKE 1666,80
9 REM .......NORMALLY POKE 1666,40
10 REM ......AFTER "SYSTEM RESET" ENTER PROG
   RAM BY "GOTO START".
11 REM
80 GOSUB 800:REM ....LOAD MACHINE PROGRAM FR
   OM "DATA"
90 START=100
100 GOSUB 700:REM ........ENTER SCROLL ROUTI
    NE
110 CONS=53279:REM .......READS CONSOLE SWIT
    CHES
120 POKE X1,PADDLE(0):POKE X2,PADDLE(1)
130 IF PEEK(CONS)<>5 THEN GOTO 120:REM LOOP
    UNTIL "SELECT" IS PUSHED
140 POKE 1677,144:REM ....STOP SCROLL MOVEME
    NT
150 IF PEEK(CONS)<>6 THEN GOTO 150:REM ..WAI
    T UNTIL "START" IS PUSHED
160 POKE 1677,208:REM ....CONTINUE SCROLLING
170 GOTO 120
697 REM
698 REM ***** GOSUB 700 STARTS SCROLLING ***
    **
699 REM
700 GRAPHICS 7+16:POKE 559,0:REM .....TV OFF
709 REM ...SET COLOR FOR BACKGROUND,TRACE X1
    ,TRACE X2
710 SETCOLOR 4,0,0:SETCOLOR 2,1,8:SETCOLOR 0
    ,8,4
720 PG=PEEK(106):REM .................NUMBER
```

```
    OF RAM PAGES
727 REM ........SCREEN DISPLAY STARTS AT 256
    *PG-4000
728 REM ........START WRITE 95 LINES(3800 WD
    S) FURTHER,I.E.
729 REM ........WRITESTART = 256*(PG-1)+56
730 POKE 203,56:POKE 204,PG-1:REM ....SET WR
    ITESTART
740 POKE 205,PEEK(560):POKE 206,PEEK(561)
750 KEEPL=PEEK(546):KEEPH=PEEK(547):REM ...S
    AVE PARAMETERS
760 POKE 546,135:POKE 547,6:REM ......MACHIN
    E PROGRAM START
770 POKE 559,34:RETURN :REM ..........TV ON
    AGAIN
771 REM
772 REM ***** GOSUB 780 EXITS FROM SCROLLING
     *****
773 REM
780 POKE 559,0:GRAPHICS 7+16:POKE 546,KEEPL:
    POKE 547,KEEPH
790 GRAPHICS 0:RETURN
797 REM
798 REM ***** GOSUB 800 LOADS MACHINE PROGRA
    M *****
799 REM
800 X1=1664:X2=1665:REM ..............PEN CO
    ORDINATES
810 DATA 200,200,40,192,48,12,3,216,165,20,4
    1,0,24,208,66,165
811 DATA 203,109,130,6,133,203,144,26,165,20
    4,105,0,41,15,72,13
812 DATA 175,6,133,204,104,208,11,174,130,6,
    169,0,202,157,0,112
813 DATA 208,250,172,130,6,169,0,136,145,203
    ,208,251,160,4,177,205
814 DATA 109,130,6,145,205,200,177,205,105,0
    ,41,15,109,175,6,145
815 DATA 205,169,255,141,250,6,173,128,6,32,
    234,6,169,85,141,250
816 DATA 6,173,129,6,32,234,6,76,95,228,201,
    180,176,17,72,74
817 DATA 74,168,104,41,3,170,189,131,6,41,25
    5,17,203,145,203,96
820 POKE 559,0
830 FOR N=1664 TO 1791
840 READ DAT:POKE N,DAT:NEXT N
850 POKE 1711,PEEK(106)-16
860 POKE 559,34:RETURN
```

**PROGRAM 3. Atari Screen As Strip Chart Recorder.**

```
0 REM ..***** MERGE WITH LISTING 1 *****
1 REM ..AT "RUN" WAITING TIME WITH BLANKED S
  CREEN, WHILE DATA
2 REM ..ARE PREPARED. SUBSEQUENT RUNS BEGIN
  AT START=100.
3 REM ..TO CHANGE PARAMETERS PRESS "SYSTEM R
  ESET", POKE IN NEW
4 REM ..VALUES AND TYPE "GOTO START" [RET].
5 REM ..POKE 1675,SPEED WITH SPEED = 0,1(PRE
  -SET),3,7
6 REM ..GIVES DIFFERENT SCROLL SPEEDS.
7 REM ..AT SPEED=0 WE GET "MULTIPLE EXPOSURE
  " DISPLAYS BY
8 REM ..POKE 1666,SKIP WITH SKIP = 30,50, OR
   70. THE NORMAL
9 REM ..VALUE IS SKIP=40 WITH DISPLAY WINDOW
   ADVANCE STEPS
10 REM ..OF 40 WORDS = 1 LINE.
11 REM
50 GOSUB 800:REM .......LOAD MACHINE PROGRAM
60 GOSUB 900:REM .......INITIAL SWING PROGRA
   M
70 POKE X2,80:REM ......SET CENTER LINE
80 POKE 1675,1:REM .....SET SPEED 1
90 START=100
100 ? :? :? :? :? "PRESS START FOR NEXT RUN"
110 IF PEEK(53279)<>6 THEN GOTO 110
120 GOSUB 700:REM ......INITIATE SCROLLING
130 W=INT(PEEK(53770)/32):REM ....GET RAND.N
    UMBER 0,..,7
140 SCORE=W:PHI=T(W):REM ..PHASE ANGLE GIVES
     AMPLITUDE
150 Q=-16:TRIALS=100:REM ..TRIALS SETS RUN L
    ENGTH
160 FOR TRY=1 TO TRIALS-1:Q=-Q:S=Q+16
170 R=PEEK(53770):REM ......GET RANDOM NUMBER
180 IF PEEK(53279)=3 THEN R=256-R:REM .INVER
    T IF "OPTION" PRESSED
190 IF R>127 THEN GOTO 220
200 W=W+1:IF W=8 THEN W=7
210 GOTO 230
220 W=W-1:IF W=-1 THEN W=0
230 SCORE=SCORE+W:PHI=T(W)
240 FOR Z=32+S TO 63+S
250 POKE X1,PEEK(SB+Z-PHI)+PEEK(SB+Z+PHI-1)
260 NEXT Z:NEXT TRY
270 GOSUB 780:? :? :? :? :REM ..EXIT SCROLLI
    NG
```

```
280 PRINT "{3 SPACES}THE SCORE IS POSITIVE O
    R NEGATIVE"
290 PRINT "{3 SPACES}IF THE AVERAGE SWING AM
    PLITUDE WAS"
300 PRINT "{3 SPACES}ABOVE OR BELOW CHANCE E
    XPECTATION"
310 ? :? :? "  SCORE = ";SCORE-TRIALS*7/2
320 ? :? :? :GOTO START
697 REM
698 REM ***** GOSUB 700 STARTS SCROLLING ***
    **
699 REM
700 GRAPHICS 7+16:POKE 559,0:REM .....TV OFF
709 REM ...SET COLOR FOR BACKGROUND,TRACE X1
    ,TRACE X2
710 SETCOLOR 4,0,0:SETCOLOR 2,1,8:SETCOLOR 0
    ,8,4
720 PG=PEEK(106):REM .................NUMBER
     OF RAM PAGES
727 REM ........SCREEN DISPLAY STARTS AT 256
    *PG-4000
728 REM ........START WRITE 95 LINES(3800 WD
    S) FURTHER,I.E.
729 REM ........WRITESTART = 256*(PG-1)+56
730 POKE 203,56:POKE 204,PG-1:REM ....SET WR
    ITESTART
740 POKE 205,PEEK(560):POKE 206,PEEK(561)
750 KEEPL=PEEK(546):KEEPH=PEEK(547):REM ...S
    AVE PARAMETERS
760 POKE 546,135:POKE 547,6:REM ......MACHIN
    E PROGRAM START
770 POKE 559,34:RETURN :REM ..........TV ON
    AGAIN
771 REM
772 REM ***** GOSUB 780 EXITS FROM SCROLLING
     *****
773 REM
780 POKE 559,0:GRAPHICS 7+16:POKE 546,KEEPL:
    POKE 547,KEEPH
790 GRAPHICS 0:RETURN
797 REM
798 REM ***** GOSUB 800 LOADS MACHINE PROGRA
    M *****
799 REM
800 X1=1664:X2=1665:REM ..............PEN CO
    ORDINATES
810 DATA 200,200,40,192,48,12,3,216,165,20,4
    1,0,24,208,66,165
811 DATA 203,109,130,6,133,203,144,26,165,20
    4,105,0,41,15,72,13
```

```
812 DATA 175,6,133,204,104,208,11,174,130,6,
    169,0,202,157,0,112
813 DATA 208,250,172,130,6,169,0,136,145,203
    ,208,251,160,4,177,205
814 DATA 109,130,6,145,205,200,177,205,105,0
    ,41,15,109,175,6,145
815 DATA 205,169,255,141,250,6,173,128,6,32,
    234,6,169,85,141,250
816 DATA 6,173,129,6,32,234,6,76,95,228,201,
    180,176,17,72,74
817 DATA 74,168,104,41,3,170,189,131,6,41,25
    5,17,203,145,203,96
820 POKE 559,0
830 FOR N=1664 TO 1791
840 READ DAT:POKE N,DAT:NEXT N
850 POKE 1711,PEEK(106)-16
860 POKE 559,34:RETURN
900 DIM T(8):REM ..GOSUB 900 INITIALS SWING
    PROGRAM
910 POKE 559,0:REM ......TV OFF
920 DEG :SB=1536:REM ....SINE TABLE STORE ON
     PAGE 6
930 FOR N=0 TO 127
940 POKE SB+N,40*SIN(N*45/8)+40:NEXT N
950 T(0)=16:T(1)=15:T(2)=14:T(3)=13
960 T(4)=11:T(5)=9:T(6)=6:T(7)=0
970 POKE 559,34:REM .....TV ON
980 RETURN
```

# Fast Banner

Sol Guber

*Run an advertisement or display banner across the screen.*

The internal registers of the Atari are easily accessed and many interesting effects are possible without much difficulty. This program will place a message on the screen and then move it along the screen, shifting the color every time the message is printed. The speed can be controlled so that the letters move very slowly or at a speed where they cannot be read. This is one step away from animation for the Atari.

The program is divided into three parts. The first part determines the message and translates it into a string variable. The second part decides which part of the string to put onto the screen. The second part consists of machine language subroutine that specifically moves the string onto the screen. The nice thing about this program is the size. The total memory usage of the program is less than 3.5K and will produce any message up to 120 characters long.

A small amount of information about the internals of the Atari is needed to understand fully how the program works. The screen memory is a continuous section of memory that stores the information linearly. For each Graphics mode, the exact location of the beginning of memory is stored in locations 88 and 89, with 89 being the HI portion of memory. The size of the screen memory depends on the Graphics mode and for Graphics 4 is 480 bytes long (10*48). Graphics 4 is a single color mode so that the information stored in each byte will determine which pixel on the screen is lit. Thus, if the byte contains the number 170 (10101010 in binary), every other pixel will be lit. If the byte contains the number 255 (11111111 in binary), every pixel will be lit. If the byte contains the number 0, then no pixel will be lit. Every spot on the screen will have a corresponding number in the screen memory, and every time a number is POKEd into the screen memory, it will have an effect on the screen.

To put a letter on the screen, the right information needs to be put into the screen memory. Because of the linearity of the screen and the fact that there are eight lines that make up each letter, it is not possible to put the eight bytes in a simple configuration in Graphics 4. To make a letter, information must be put in a screen memory byte, then in a screen memory ten bytes further, then in a screen memory ten bytes further, and so forth. The information on how to make the letter is its binary component, and this is made up of eight bytes. This information is presently stored in memory and can be generated for each letter as needed. To use this information, it must be removed from memory and stored in a variable. The two methods to do this would be a dimensioned variable or a string variable for the large amounts of information needed. The Atari BASIC stores each number as a six byte word and takes up a great deal of memory to store many numbers. The Atari BASIC stores string variables with one letter per byte and is much more compact when a great deal of information needs to be stored. Thus, the eight bytes of each letter are stored as a part of a string, and the 960 numbers take up only 960 bytes rather than the 5760 that would be taken up if they were stored as numbers.

A description of the program follows.

Lines 5-7 initialize several constants that will be used throughout the program. Line 10 dimensions the three string variables that will be used. Line 15 puts the value of 0 into the string L$ and line 18 determines where in memory the string L$ is found. Lines 20-48 determine the message, its length, how many times it is to cycle, and the speed of it.

Lines 50-120 translate the string message into its components and put it into the string L$. The subroutine at 600 determines where the offset of the letter is found in memory, and variable 13 contains that value. The value in that byte is stored in L$ with a 120 byte offset for each line. The sound is turned on to show that something is happening. It cycles through eight times for each letter, and each letter is put into an appropriate spot in the string L$.

Line 125 turns off the sound and goes into Graphics 4. Line 127 goes to the beginning of the string L$ and defines a variable R which will be the color variable. In Graphics 4, the color register is located at 708 and the value in 708 defines the color of the letters. Its format is that the first four bits in the byte define the intensity of the color and the last four bits define the color itself, with the first bit in the byte being ignored. Thus, there are eight intensities to the color and 16 colors possible. To shift from one color to the next at the same intensity, a multiple of 16 is added to the value of register 708. The maximum value is 255 as for any register. Line 129 defines the point on the screen memory where the message will start.

Lines 130 to 190 put the message into the screen memory and the system itself puts it onto the screen. Line 130 defines the number of cycles for the message. Lines 133-134 define the color for the message cycle. The heart of the movement is in the loop from 140-165. Line 153 points to the letter in the message that is to be on the far left of the screen. The machine language subroutine on page six (1536) moves the appropriate letters to the screen memory. Line 163 is a timer that displays the line for a certain amount of time before it is shifted over to the next letter. When the speed is fast, the time lag is short and the letters look as if they were marching across the screen. When the speed is very fast, the letters are a blur. The subroutine at 3000 reads in a machine language subroutine into page six of memory where it is used to move parts of the string variable L$ into the screen memory.

This program opens up several possibilities for further utilization. The subroutine can be modified to put anything on the screen anywhere. With changes in the graphics mode, the size of the letters can be doubled or decreased by 25%. Any of the graphics characters can be used by the program. With modification, the letters can be reversed or made upside-down or even sideways. Using interrupts, the colors can flow along with the letters so that each letter can be a different color. The possibilities are endless for showing off the capabilities of your Atari computer.

**PROGRAM. Fast Banner.**

```
5 C1=1:C16=16:C4=4:C0=0:C7=7
6 LTT=600:C31=31:C32=32:C128=128:C96=96:C64=
  64:C127=127:C12=C4+C7+C1
7 C2=2:C120=120:C60=C64-C4
10 DIM L$(960),A$(110),O$(1)
15 L$(1)="{,}":L$(960)="{,}":L$(2)=L$
18 LPT=ADR(L$)+C4+C1
20 ? "WHAT IS YOUR MESSAGE"
30 INPUT A$:N=LEN(A$)
40 ? "HOW MANY CYCLES":INPUT N1
45 GOSUB 3000
48 ? "HOW FAST 1-SLOW  10-FAST":INPUT S2:S2=
   (C128+C32)/S2
50 FOR I=C1 TO N
60 O$=A$(I,I):GOSUB LTT
70 I3=57344+X*8:FOR J=C0 TO C7:M=PEEK(I3+J)
80 Y=J*120+I:POKE Y+LPT,M
115 SOUND 1,M+C16,10,8
120 NEXT J:NEXT I
125 SOUND 1,0,0,0:GRAPHICS C4+C16
127 LPT=ADR(L$):R=C16-C4
129 T=PEEK(88)+256*PEEK(89)+59
130 FOR K=C1 TO N1
133 R=R+C16:IF R<=255 THEN POKE 708,R:GOTO 140
134 R=12:POKE 708,R
140 FOR I=C0 TO N+C4
155 A=USR(1536,LPT+I,T)
163 FOR K1=1 TO S2:NEXT K1
165 NEXT I
175 R=PEEK(708)
190 NEXT K
500 STOP
600 X=ASC(O$):IF X>C127 THEN X=X-C128
610 IF X>C31 AND X<C96 THEN X=X-C32:RETURN
620 IF X<C32 THEN X=X+C64
630 RETURN
3000 FOR I=1536 TO 1600:READ X:POKE I,X:SOUN
     D 1,X,10,10:NEXT I
3005 SOUND 1,0,0,0
3010 DATA 104,104,133,199,104,133,198,104,133
3020 DATA 201,104,133,200,162,8,160,12
3030 DATA 177,198,145,200,136,208,249,24
3040 DATA 165,200,105,10,133,200,144,3
3050 DATA 230,201,24,165,198,105,120,133,198
3060 DATA 144,3,230,199,24,202,208
3070 DATA 221,96,0,0,0,0,0,0,0,0
3080 DATA 202,208,215,96,0,0,0,0,0,,24
3090 RETURN
```

# Perfect Pitch

Fred Coffey

"It sounds a bit out of tune," said my musician friend after listening to my Atari's rendition of "The Star Spangled Banner" in two voices. "You should work on pitch control."

"It sounds fine to me the way it is," I protested. "Anyway, let's play Asteroids."

Later I began to wonder just how good the Atari's pitch was. I remembered that the note "A" above "middle C" was usually tuned to 440 cycles per second. And the *Atari BASIC Manual* said that if I put a "72" in for the pitch control (P) in the command "SOUND 0,P,D,V", I would get the note "A." Was this sound actually 440 Hz?

So I turned to my newly acquired *Atari Hardware Manual* for help. I finally determined that the Atari controls pitch by dividing an internal 64 kilohertz clock according to the following formula (where "P" is an integer between 0 and 255):

$$\text{PITCH} = 63921.0/(2^*(\text{P}+1)) \qquad (1)$$

So, if we plug the "72" (which Atari says is our "A" note) into the above equation, we get a pitch of 437.8 Hz. The Atari *is* out of tune! It is, in fact, not possible to generate a precise "440" note.

But the *Atari Hardware Manual* does have a solution – we are allowed to switch to 16-bit precision on our note generation as long as we are willing to settle for two voices instead of four.

However, before we can exploit that, we need to learn a new way of controlling sound. We will have to POKE instructions instead of using the BASIC SOUND command. It's really no problem since we can define an exact equivalence to SOUND as follows:

**Table 1**

| Voice | SOUND Command | Equivalent POKE |
|---|---|---|
| 1 | SOUND 0,P,D,V | POKE 53760,P: POKE 53761,(16*D)+V |
| 2 | SOUND 1,P,D,V | POKE 53762,P: POKE 53763,(16*D)+V |
| 3 | SOUND 2,P,D,V | POKE 53764,P: POKE 53765,(16*D)+V |
| 4 | SOUND 3,P,D,V | POKE 53766,P: POKE 53767,(16*D)+V |

Now we can go on. The Atari has a memory register, at location 53768, which has 256 ways (i.e., one byte) to control sound options. I don't understand most of them yet, but I did sort out a few of them. Just as an example I found that if I "POKE 53768,1" the Atari switches to a 15 kilohertz clock for sound control, and equation (1) above becomes:

PITCH = 15699.9/(2*(P+1)) (2)

But to get back to our immediate objective of more precise pitch control, I found that if I do a "POKE 53768,80" then voices one and two merge into a single high-resolution voice. If I "POKE 53768,120" voices three and four follow suit.

We can control these new combined voices as follows:

Voice 1+2

```
POKE 53760,P1: POKE 53762,P2: POKE 53763, (16*D)+V
```

Voice 3+4

```
POKE 53764,P1: POKE 53766,P2: POKE 53767, (16*D)+V
```

You will recognize the distortion (D) and volume (V) terms from the old SOUND commands. But what are these new terms P1 and P2? They are simply a pair of integer sound control terms like the "P" in equations (1) and (2), and they generate a high-precision sound using a 1.79 megahertz clock as follows:

PITCH = 1789790/(2*(256*P2+P1+7)) (3)

Conversely, if we want to know what values to POKE in order to generate an objective pitch, we can solve (or let the computer solve) the following and POKE the values into the computer:

P2 = INT ((1789790/(2*PITCH)-7)/256) (4)
P1 = INT (1789790/(2*PITCH)-7 -256*P2+.5) (5)

To generate a 440 Hz note, then, we can solve the above equations to find that P2=7 and P1=235. We can then POKE these values to generate our note (which comes out at 439.97 Hz – pretty good!).

So much for the mathematics. Now for a demonstration (Program 1). The demonstration uses two joysticks to control one high-resolution voice and one "normal" voice, and displays the relevant pitch equations on the screen as the equation terms change. You can quickly see how much control over pitch you have in each case.

Program 2 allows you to experiment with different scales of notes – using the best normal Atari approximation and then playing the scale again with high resolution pitch control.

Finally, I was ready to call my musician friend back in. "Listen to this," I said. "I can control pitch to a fraction of a cycle per second. In fact, if I simultaneously sound two notes that are very close to each other, you can hear the combined sound waver as the two wave forms drift in and out of phase and alternately reinforce and cancel each other."

"Not bad," he said. "We call that phenomenon 'beating' and use it to tune instruments precisely against a tuning fork reference."

"Hey," I said, "that means you could play a note on your piano and I could match my Atari note to it precisely by listening to the beat. I've invented a piano tuner!

"Now," I said, "if you'll just tell me the mathematical relationship between the rest of the notes on the musical scale, I'll tune my Atari and we'll hear some real music!"

"Well," he said, "it's not that simple. There are several methods of tuning. It depends on what instrument you're tuning, and even depends on what country you live in. Besides, pitch control isn't the only problem with your Atari sound. For example..."

"It sounds fine to me the way it is," I said. "Let's play Asteroids."

**PROGRAM 1. Perfect Pitch**

```
1 REM --------  PROGRAM 1  --------
2 REM
3 REM
4 GRAPHICS 0:? :? :? :? "WHAT FREQUENCY WOUL
  D YOU LIKE TO":? "START";:INPUT FREQ
5 AUDF3=INT(63921/(2*FREQ)-1+0.5)
6 AUDF2=INT((1789790/(2*FREQ)-7)/256)
7 AUDF1=INT((1789790/(2*FREQ))-7-256*AUDF2+0.5)
8 IF FREQ>125 AND FREQ<16000 THEN 10
9 ? "ALLOWABLE RANGE 125 TO 16000":? "TRY AG
  AIN";:INPUT FREQ:GOTO 5
10 GRAPHICS 0:? "JOYSTICK #1 CONTROLS HIGH R
   ESOLUTION"
15 ? "FREQUENCY.  PRESS BUTTON TO SOUND,"
20 ? "MOVE UP/DOWN FOR SLOW FREQUENCY SHIFT"
   :? "OR LEFT/RIGHT FOR FAST SHIFT."
30 ?:?"JOYSTICK #2 CONTROLS LOW RESOLUTION,"
40 ?"BUTTON TO SOUND OR UP/DOWN FOR SHIFT.":?
50 REM FOR UNKNOWN REASON FOLLOWING COMMAND
   NECESSARY FOR RELIABLE OPERATION:
55 SOUND 0,0,0,0:SOUND 1,0,0,0:SOUND 2,0,0,0
   :SOUND 3,0,0,0
100 POKE 752,1
110 POKE 53768,80
130 POKE 53763,10*16+15
140 POKE 53765,10*16+15
160 IF STICK(0)=13 THEN AUDF1=AUDF1+1:IF AUD
    F1>255 THEN AUDF2=AUDF2+1:AUDF1=0
170 IF STICK(0)=14 THEN AUDF1=AUDF1-1:IF AUD
    F1<0 THEN AUDF2=AUDF2-1:AUDF1=255
180 IF STICK(0)=11 THEN AUDF2=AUDF2+1
190 IF STICK(0)=7 THEN AUDF2=AUDF2-1
220 IF STICK(1)=13 THEN AUDF3=AUDF3+1
230 IF STICK(1)=14 THEN AUDF3=AUDF3-1
240 IF STRIG(0)=0 THEN POKE 53760,AUDF1:POKE
     53762,AUDF2:GOTO 260
250 POKE 53760,0:POKE 53762,0
260 IF STRIG(1)=0 THEN POKE 53764,AUDF3:GOTO280
270 POKE 53764,0
280 FOUT2=1789790/(2*(AUDF2*256+AUDF1+7))
290 FOUT3=63921/(2*(AUDF3+1))
300 POSITION 1,9:?
310 ? "HI RES FREQUENCY:":?
320 ? " FREQ = 1789790/(2*(";AUDF2;"*256+";A
    UDF1;"+7)){3 SPACES}":?
325 ? "{7 SPACES}= ";FOUT2
330 ? :? :? "LO RES FREQUENCY:":?
340 ? " FREQ = 63921/(2*(";AUDF3;"+1))
    {5 SPACES}":? :? "{7 SPACES}= ";FOUT3
500 GOTO 100
```

**PROGRAM 2. Perfect Pitch.**

```
2 REM
5 REM PROGRAM 2
10 REM
20 REM THE FOLLOWING PROGRAM TAKES A GIVEN S
   CHEDULE OF EIGHT PITCH VALUES
30 REM (I.E. A SCALE) AND PLAYS THE NOTES US
   ING THE BEST NORMAL ATARI
40 REM APPROXIMATION AND THEN USING A "HIGH
   RESOLUTION" RENDITION
50 REM
60 REM
70 REM
90 REM "EQUAL TEMPERMENT" SCALE
100 DATA 520,584,655,694,779,874,982,1040
110 DIM PITCH(8),A$(1)
120 FOR J=1 TO 8:READ X:PITCH(J)=X:NEXT J
150 GRAPHICS 0:? :? " NORMAL ATARI
    {7 SPACES}HIGH RESOLUTION":? " ---------
    ---{7 SPACES}----------------"
160 ? "  P{4 SPACES}PITCH{8 SPACES}P1  P2
    {3 SPACES}PITCH":? " --{4 SPACES}-----
    {8 SPACES}--  --{3 SPACES}-----"
170 POKE 752,1
200 REM GENERATE 'NORMAL' ATARI SOUND
210 FOR J=1 TO 8
220 P=INT(63921/(2*PITCH(J))-1)
230 SOUND 2,P,10,8
240 PITCH=63921/(2*(P+1))
250 POSITION 3,J+6:? P;"  ";PITCH
260 FOR W=1 TO 200:NEXT W:NEXT J
270 SOUND 2,0,0,0:FOR W=1 TO 500:NEXT W
300 REM 'HIGH RESOLUTION' SOUND
305 POKE 53768,80
310 FOR J=1 TO 8
320 P2=INT((1789790/(2*PITCH(J))-7)/256)
330 P1=INT(1789790/(2*PITCH(J))-7-256*P2+0.5)
340 POKE 53760,P1:POKE 53762,P2:POKE 53763,(
    16*10)+8
350 PITCH=1789790/(2*(256*P2+P1+7))
360 POSITION 21,J+6:? P1:POSITION 25,J+6:? P
    2;" ";PITCH
370 FOR W=1 TO 150:NEXT W:NEXT J
380 POKE 53760,0:POKE 53762,0
400 POSITION 2,19:? " PLAY IT AGAIN (Y OR N)
    ";:INPUT A$:IF A$="Y" THEN 150
410 ? "ENTER 8 NEW PITCH VALUES, ONE AT A":?
     "TIME"
420 FOR J=1 TO 8:INPUT X:PITCH(J)=X:NEXT J:G
    OTO 150
```

CHAPTER FIVE

# BEYOND BASIC

# Put Your USR Code Into A BASIC Program Automatically

F. T. Meiere

*Entering machine language into a BASIC program can be tedious, but with AUTOTYPE, you just enter the file name and a BASIC subroutine is automatically written for you.*

This utility routine automatically reads machine language code into your BASIC program as a graphic string. It DIMensions the string properly and successfully handles the troublesome quote and carriage return. All you type is RUN. If your fancy turns to READ and POKE, then minor changes will put the code in DATA statements. None of the ideas are new, but the key step is POKE 842,13 from **COMPUTE!**, August 1981, #15. My thanks to our sharp-eyed editor who pointed out the potential of this POKE which has been around for some time (e.g., Santa Cruz Software "Memory Map for the Atari"). For entertainment and debugging, you may want to skip the input of actual code, read in the integers from 1 to 255, and watch your Atari program itself.

To enter a USR program as a string, create relocatable machine code and save it to disk or cassette. ENTER this program AUTOTYPE temporarily in your BASIC program. GOTO 9000. The proper string will be entered and displayed on the screen. The whole program can be SAVEd or just the desired portion LISTed to disk or cassette. Alternatively, AUTOTYPE can be RUN separately and the string LISTed and ENTERed as needed.

**Line 9000-9310:** The disk file containing USR code is OPENed and the first six bytes of DOS information are removed. Cassette users will OPEN #C1,4,0,"C:". The remainder of the section makes the program user friendly by setting default values and error messages. The program can be shortened by omitting this whole section, providing the USR file is opened and values given to NAME$,LN1 and INCR.

**Line 10020:**PDIM dimensions the string.

**Line 10030:**LNUM prints the line number and the string name. XPR is the position of the first string entry on the current line.

**Line 10050:**PSPEC enters the quote and carriage return (EOL) separately using CHR$. Lines 10410 and 10420 make sure there is information to enter and remove the quote mark which would normally follow the line number.

**Line 10060:**CHR$(27) = ESC ensures that the control characters will print. If you POKE 766,1 to do this, then LNUM cannot clear the screen.

**Line 10070:** The final quote is added and LENT enters the whole line into this program.

**Line 10220:**POKE 842,13 makes the Atari shoot carriage returns continuously after the STOP. This enters the string as printed on the screen. However, when the CONT printed by line 10120 is encountered, the program will resume with line 10230 POKE 842,12 which returns the computer to normal.

If the USR code is not relocatable, you may want to enter it as DATA and POKE it into memory. The following changes will enter the code as DATA statements.

```
9130 TRAP 40000
10060 ? X;",";:IF PEEK(84)=4 THEN IF PEEK(85
      )>30 THEN GOSUB LENT
10070 NEXT I:GOSUB LENT
10110 ? CHR$(125):?:? L;" DATA ";:XPR=PEEK(8
      5)
10205 ? CHR$(126)
```

Omit the references to PDIM and PSPEC on Line 10020, and omit Line 10050 and Lines 10300 to the end.

One final comment: There are at least three ways to save USR code as part of a BASIC program. 1) Put it in as a SUBR$ string and call USR(ADR(SUBR$)). 2) Put it in DATA statements and POKE it into RAM at a fixed address. 3) Load it immediately following your program and change the pointers to fool BASIC into saving your code along with the program. BASIC does not normally save string space and option 3 can be rather tricky. However, many descriptions, including those in the Atari manuals, are misleading. I recommend that you consider your specific needs and take any other program (including this one) with a grain of salt.

## PROGRAM. Put Your USR Code Into A BASIC Program Automatically.

```
9000 REM Convert USR code to a string
9010 AUTOTYPE=10000:XYZ=9200
9020 DIM FILE$(15),NAME$(15):C0=0:C1=1:C2=C1
     +C1
9030 GRAPHICS 0:POSITION 10,C1:? " AUTOTYPE "
     ":? :? " Convert USR code to a string "
9040 ? :? "Please enter information below":?
      "For default values (..) hit RETURN"
9050 ? :? "USR code FILE NAME ";:INPUT NAME$
     :IF NAME$="" THEN 9050
9060 FILE$=NAME$:TRAP 9070:IF NAME$(1,1)="D"
      THEN IF (NAME$(2,2)=":" OR NAME$(3,3)=
     ":") THEN 9080
9070 FILE$(1,2)="D:":FILE$(3)=NAME$
9080 TRAP 9300:OPEN #C1,4,0,FILE$:GOSUB XYZ
9090 IF (X<>255 OR Y<>255) THEN ? "NOT LOAD
     FILE",FILE$:GOTO 9310
9100 GOSUB XYZ:L=Z:GOSUB XYZ:BYTES=Z-L+C1
9110 LN1=100:TRAP 9120:? "First line number
     (100) ";:INPUT LN1
9120 INCR=10:TRAP 9130:? "Incr. line number
      (10) ";:INPUT INCR
9130 TRAP 40000:? "USR string name (SUBR$) "
     ;:INPUT NAME$:L=LEN(NAME$):IF L<2 THEN
     NAME$="SUBR$":L=5
9140 NAME$(L)="$"
9150 GOSUB AUTOTYPE:CLOSE #C1:END
9200 REM XYZ=get X&Y,calculate Z
9210 GET #C1,X:GET #C1,Y:Z=X+256*Y:RETURN
9300 ? " NO FILE NAMED ",FILE$
9310 ? :? "Hit RETURN to RUN again ";:INPUT
     NAME$:RUN
10000 REM {3 SPACES}AUTOTYPE{3 SPACES}
10010 REM Type lines in program
10020 LNUM=10100:LENT=10200:PDIM=10300:PSPEC
      =10400:L=LN1:GOSUB PDIM
10030 K=C0:FOR I=C1 TO BYTES:IF K=C0 THEN GO
      SUB LNUM
10040 GET #C1,X:K=K+C1:REM code byte
10050 IF (X=34 OR X=155) THEN GOSUB PSPEC:GO
      TO 10070
10060 ? CHR$(27);CHR$(X);:IF K=80 THEN ? CHR
      $(34):GOSUB LENT
10070 NEXT I:IF K<>0 THEN ? CHR$(34):GOSUB L
      ENT
10080 GRAPHICS C0:LIST C1,8999:RETURN
10100 REM LNUM=print line # and incr
```

```
10110 ? CHR$(125):? :? L;" ";NAME$;"(";I;")=
      ";CHR$(34);:XPR=PEEK(85)
10120 POSITION C2,6:? "CONT":POSITION XPR,2:
      RETURN
10200 REM LENT-enter line
10210 ? CHR$(126):POSITION C0,C0:K=C0:L=L+IN
      CR
10220 POKE 842,13:STOP :REM auto <CR>
10230 POKE 842,12:REM stop auto <CR>
10240 RETURN
10300 REM PDIM-print DIM statement
10310 ? CHR$(125):? :? L;" DIM ";NAME$;"(";B
      YTES;")"
10320 POSITION C2,6:? "CONT":GOSUB LENT:RETU
      RN
10400 REM [illegible]-quote and <CR>
10410 IF PEEK(85)<>XPR THEN ? CHR$(34):GOSUB
       LENT:GOSUB LNUM
10420 ? CHR$(30);"CHR$(";X;")":GOSUB LENT:RE
      TURN
```

# Back Up Machine Language Programs With BASIC

Ed Stewart

*This fairly technical article shows you how to back up cassette-based machine language programs. If that's not a priority for you, it's still worthwhile reading. The author explains IRGs (interrecord gaps) on the Atari and a good bit more.*

If you have any machine language programs on cassette tape, you may be painfully aware of what a "BOOT ERROR" message means to you. If you haven't yet experienced the anguish of a non-readable machine language tape, then read on and avoid future pain. For those of you who are not masochistic and who, like me, have lost a tape or two, take heart, for you can now save a backup copy in your wine cellar tape vault. The program described in this article allows you to make your own private copy of any Atari machine language program. It is almost as easy as CLOAD/CSAVE and is well worth your time to incorporate into your program library.

When I began developing this BASIC program, I thought it would be easy as GET/PUT and would be a trivial program requiring perhaps 30 minutes at the outside to develop. Wrong. After about 16 hours of work over three weeks I came up with this solution. I could not have done it without the Atari Operating System source listing. My final version of the program ended up with two machine language subroutines filling and emptying a very long string variable while tricking the Operating System Cassette Handler into doing all the I/O for me.

The basic reason why I had to resort to machine language was very simple – BASIC was too slow to do the job. Each block of data stored on a cassette tape is 128 bytes in length. All I had to do was read a block of data from the tape, transfer all 128 bytes to my string, and read the next block of data in before the tape ground to a halt. Each block of data on the tape is separated by a gap called an interrecord gap (IRG). There are two kinds – short ones and long ones. The short

IRG's are used when you can dispose of the 128-byte data block quickly and request the next one while the record motor is still on. If the recorder stops between blocks on a tape that has short IRG's, then when it starts back up again it will begin reading from the tape just a little bit beyond where the data really is. The result is usually error code 143 – Serial Bus Data Frame Checksum Error. Long IRG's, on the other hand, are long enough to permit the tape to come to a complete stop and start again without any loss of data or error codes. Long IRG's are used, therefore, when the 128-byte data block in the cassette buffer cannot be used very fast, for example, when you use the GET commands in BASIC. Machine language programs are stored on tape with short IRG's. By the time you can issue 128 GET commands and transfer those 128 bytes into a string, the tape has stopped. The 129th requested GET will recognize an empty cassette buffer and cause a new data block to be read from the tape, but it will be read too late – the motor stopped with a short IRG tape and bingo – error 143.

The solution was to request a data block from tape with a GET command, empty the cassette buffer into my string with a machine language subroutine and make the machine think that its cassette buffer was empty so that a subsequent GET would cause another data block to be read into the buffer. This process was repeated until the EOF condition was obtained on the tape. Then the string had to be written back to tape, using another machine language program to empty the string into the cassette buffer while using the PUT command to cause the actual tape write to occur. If this seems a bit complex to you, perhaps the following diagram will help.

| READ SIDE | WRITE SIDE |
|---|---|
| Open | Open |
| Basic "GET" | empty string to cassette |
| Move cassette buffer to string | mark cassette buffer full |
| mark cassette buffer empty | Basic "PUT" |
| EOF – CLOSE | EOF – CLOSE |

The only limitation this program has is that the program size you may copy is limited by the size of the string A$. The size of the string A$ is limited by the size of your available RAM and is derived dynamically based upon your RAM size. A string cannot exceed 32K and so A$ is limited to 32K. In other words, a program greater than 32K cannot be copied.

Now that I have showered you with all the technicalities of this little program, let's see what it looks like line by line.

**PROGRAM. Back Up Machine Language Programs With BASIC.**

```
1 REM BACKUP TAPE UTILITY FOR MACHINE
2 REM LANGUAGE PROGRAM OR TO BACKUP
3 REM ANY 600 BAUD TAPE WITH SHORT
4 REM IRG'S FOR THAT MATTER
5 REM
6 REM AUTHOR ED STEWART
10 NO=0:N1=1:N2=2:N256=256:GRAPHICS N2+16:RE
   M SET LOW MEMORY GRAPHICS MODE
20 Z=PEEK(742)*N256+PEEK(741)-PEEK(145)*N256
   +PEEK(144)-1500:IF Z>32767 THEN Z=32767
24 DIM A$(Z):REM SET STRING LENGTH
30 A$(1)="{,}":A$(Z)="{,}":A$(2)=A$
34 REM INITIALIZE STRING IN 30
40 POKE 203,ADR(A$)-(INT(ADR(A$)/N256)*N256)
   :POKE 204,INT(ADR(A$)/N256):REM POKE STR
   ADR FOR M.L. ROUTINE
60 FOR I=1536 TO 1565:READ A:POKE I,A:NEXT I
   :REM POKE IN M.L. ROUTINE
70 TRAP 200:REM SET TRAP FOR EOF
74 ? #6;"INSERT INPUT TAPE":? #6;"PRESS RETU
   RN KEY{4 SPACES}BEGIN"
80 OPEN #N1,4,255,"C":CNT=NO:REM OPEN INPUT
   FILE
90 FOR I=NO TO Z STEP 128:REM SET INPUT LOOP
    COUNTER
100 GET #1,B:CNT=CNT+128:REM FILL CASSETTE B
    UFFER FROM TAPE
120 X=USR(1536):REM MOVE BUFFER TO STRING AN
    D MARK BUFFER EMPTY
140 NEXT I:? "NOT ENOUGH RAM TO COPY TAPE":E
    ND
200 IF PEEK(195)=136 THEN CLOSE #N1:GRAPHICS
     N2+16:? #6;"INSERT OUTPUT TAPE"
202 ? #6;"PRESS RETURN KEY TO{5 SPACES}BEGIN":G
    OTO 210
204 ? "ERROR - ";PEEK(195):END
210 RESTORE 10000:REM SETUP FOR 2ND M.L. PRO
    GRAM
220 FOR I=1536 TO 1566
230 READ B:POKE I,B:NEXT I:REM POKE IN 2ND P
    ROGRAM
234 POKE 203,ADR(A$)-(INT(ADR(A$)/N256)*N256
    ):POKE 204,INT(ADR(A$)/N256):REM SET UP
    STRING ADD FOR 2ND PGM
240 OPEN #N1,8,255,"C":REM OPEN OUTPUT TAPE
260 FOR I=NO TO CNT STEP 128:REM SETUP OUTTA
    PE LOOP COUNTER
```

```
262 X=USR(1536):REM EMPTY STRING TO CASSETTE
     BUFFER AND MARK BUFFER FULL
270 Z=ASC(A$(I+128)):PUT #N1,Z:REM PUT LAST
    BYTE IN BUFFER AND WRITE TO TAPE
300 NEXT I
320 CLOSE #N1:GRAPHICS N2+16:? #6;"THAT'S AL
    L FOLKS":REM SAY DONE OK NOW
400 FOR I=NO TO 800:NEXT I:RUN :REM MAKE MOR
    E OUTPUT TAPES IF DESIRED
9000 DATA 104,174,138,2,134,61,160,0,162,0,1
     85,0,4,129,203,200,230,203,208,2,230,20
     4,196,61,240,3,76,10,6,96
10000 DATA 104,169,128,133,61,160,0,162,0,16
      1,203,153,0,4,200,230,203,208,2,230,20
      4,196,61,240,3,76,9,6,198,61,96
```

# Loading Binary DOS Files From BASIC

Robert E. Alleger

*You can load binary (machine language) files from DOS with selection "L." Here's a machine language program that lets you do it from BASIC.*

## Introduction

Several months ago, my friend Doug came to me and said, "Hey Bob, I want to show you the nifty menu program I wrote." After he demonstrated his program, I said, "Big deal! You came all the way over here to show me this?" He replied, "Not exactly. There's one slight problem that I don't know how to solve. My menu will not load machine language programs. I thought that you might be able to help me." After some arm twisting, I agreed to write a routine that would allow a BASIC program to load machine language disk files.

## DOS 2 LOAD File Format

Before I begin with a step-by-step breakdown of LOADIT, it might be helpful to define the format of a DOS 2 binary load file (see Figure 1). A binary load file begins with a two-byte header id of $FF $FF ($xx indicates hexadecimal numbers), followed by a two-byte start address, a two-byte end address, and the program data (object code).

For programs with multiple ORGs, this pattern may repeat over and over again, beginning with the start address. If the file was created by using the DOS copy ('C') command to append two or more files together, then the pattern may repeat beginning with the header id.

## Figure 1.

**: $FF $FF : start address : end address : object code : ...**

## On With The Show

LOADIT is designed to be called from BASIC via the USR function. It is ORG'd for page six (1536-1791) so that it is relatively safe, as long as the machine language program that it runs does not load into this area.

Referring to Program 1 (LOADIT.ASM) line numbers 0440-0580

(INIT) is the initialization routine. It calls a subroutine that CLOSEs IOCB (I/O Control Block) number one (in case it was already open), retrieves the address of the file-spec from BASIC, and then OPENs the specified file.

Lines 0620-0810 (RDHDR) read the first two bytes of a block of object code from the input file. If both bytes are an $FF (header id), then the program loops back to get the next two. Together, these bytes form the address into which to start loading the object code. An end of file error ($88) at this point indicates that the whole file has been loaded, and therefore execution branches to the DONE routine.

Lines 0850-0930 (CONT) read the next two bytes, which form the ending address for the current block of object code. Any error returned in the Y-register by CIO at this point either indicates that the file is bad (i.e., "File Number Mismatch," etc.) or that the file is not in binary load format (see Figure 1).

Lines 0970-1040 (HDROK) check to see if this is the first block of object code that has been read from this file. If it is, then the address of the first instruction is used as the default run address, in case none is specified. In assembly language, the run address is specified by storing the address of the entry point to your program in locations $02E0-02E1.

Lines 1080-1270 (RDBLOK) read the object code from the file and store it in memory, from start address through end address. The number of bytes to be read is calculated by taking the ending address, subtracting the start address, and adding one. The only non-fatal error code that CIO could return at this point is $03, which indicates that an end of file error will be encountered on the next read.

After loading the entire block of object code, the program loops back to the RDHDR routine.

Lines 1340-1350 (DONE) are executed on an end of file error in the RDHDR routine. The input file is CLOSEd and execution is transferred to the loaded program via the vector at address $02E0-02E1.

## Does It Really Work?

I will use a very simple, no frills menu program (see Program 2) to demonstrate that LOADIT really does work. Although LOADIT might be useful in other applications, I chose this menu because it illustrates the most common usage.

The subroutine starting at line number 5000 is responsible for placing LOADIT at its proper location in memory. A FOR/NEXT loop reads the decimal equivalent of the machine language instructions and POKEs them into page six of memory.

LOADIT is only called when a machine language program is chosen from the screen menu. I chose to indicate a machine language load-and-go file by using the file extension ".CMD" (from my old TRS-80 days). Of course you can use anything you like, just change line number 480.

LOADIT is called by the statement in line number 550. The parameters specified in the USR function are LOADIT's starting address (1536 = $0600) and the address of the string variable which contains a complete file-spec (i.e., Dn:name.ext).

BASIC should never fall through to line number 570, because LOADIT only returns to BASIC if an error is encountered.

## Did I Really Type All Those DATA Statements?

In case you were wondering, I did not type in the DATA statements on line numbers 5001-5008. Instead, I used a handy BASIC utility program that I wrote called DATAGEN (see Program 3). It is included as a bonus.

DATAGEN reads a binary load format file, such as LOADIT.OBJ, and produces a file that can be appended to your BASIC program with the ENTER command (see page 25 of the ATARI "BASIC REFERENCE MANUAL"). This file will contain one FOR/NEXT loop and a number of DATA statements for each block of object code in the file.

Upon startup, DATAGEN requests a complete file-spec for the input and output files. It also asks for the starting line number that will be used to begin numbering the subroutine that is being written to the output file.

## And That's Not All, Folks

Program 4 is another utility program which can be used to find out the load parameters of a binary load format file. DPSLOAD reads a binary file and displays the starting and ending address of each block of object code, the auto run, and init addresses, if present (see pages 41-44 of the Atari *Disk Operating System II Reference Manual*).

The input requested by DSPLOAD is a complete file-spec of the input file. The information will be displayed on the screen when each block of object code is encountered.

To some, it may seem that DATAGEN and DSPLOAD have nothing to do with my intended topic, but I hope that they prove to be educational as well as useful aids to programming.

**PROGRAM 1. Loading Binary DOS Files From BASIC.**

```
10  .TITLE "LOADIT  1.1  02/24/82"
20 ;Author: Robert E. Alleger
30 ;
40 ;This program allows a BASIC
50 ;program to LOAD a machine
60 ;language program and execute it.
70 ;
80 ;
90 ;
0100 IOCB1=1*16 ;IOCB #1 (D:)
0110 ;IOCB (8 * 16 bytes)
0120 ICHID=$0340 ;handler ID
0130 ICDNO=ICHID+1 ;device #
0140 ICCOM=ICDNO+1 ;command
0150 ICSTA=ICCOM+1 ;status
0160 ICBAL=ICSTA+1 ;buffer address
0170 ICPTL=ICBAL+2 ;PUT routine addr - 1
0180 ICBLL=ICPTL+2 ;buffer length
0190 ICAX1=ICBLL+2 ;AUX 1
0200 ICAX2=ICAX1+1 ;AUX 2
0210 ICAX3=ICAX2+1 ;AUX 3
0220 ICAX4=ICAX3+1 ;AUX 4
0230 ICAX5=ICAX4+1 ;AUX 5
0240 ICAX6=ICAX5+1 ;AUX 6
0250 ;
0260 CIO=$E456 ;CIO entry point
0270 ENR=$03 ;EOF on next read
0280 EOF=$88 ;EOF status
0290 OPEN=$03 ;OPEN command
0300 GETCHR=$07 ;GET CHARACTERS command
0310 CLOSE=$0C ;CLOSE command
0320 OREAD=$04 ;OPEN direction= READ
0330 ;
0340 RUNLOC=$02E0 ;auto run vector
0350 FREE0=$00CB ;free 0 page RAM (to $00D1)
0360 HEADER=FREE0 ;block header buffer
0370 FLAG=HEADER+4 ;1st block flag
0380 ;
0390 ;
0400  *=$0600
0410 ;
0420 ;Initialization
0430 ;
0440 INIT LDX #IOCB1
0450  JSR CLOSEIT ;in case IOCB was in use
0460  STX FLAG
0470  PLA ;get rid of # of args on stack
0480  PLA ;MSB of file spec location
```

```
0490  STA ICBAL+1,X
0500  PLA ;LSB
0510  STA ICBAL,X
0520  LDA #OREAD
0530  STA ICAX1,X
0540  LDA #OPEN
0550  STA ICCOM,X
0560  JSR CIO ;open file
0570  BPL RDHDR
0580  JMP ERROR ;file not found
0590 ;
0600 ;Read header id or start address
0610 ;
0620 RDHDR LDA #HEADER&255
0630  STA ICBAL,X
0640  LDA #HEADER/256
0650  STA ICBAL+1,X
0660  LDA #2
0670  STA ICBLL,X
0680  LDA #0
0690  STA ICBLL+1,X
0700  LDA #GETCHR
0710  STA ICCOM,X
0720  JSR CIO ;get id or start addr
0730  BPL CHKID
0740  CPY #EOF ;end of file?
0741  BEQ DONE ; -yes
0742  BNE ERROR ; -no, bad file
0750 ;
0760 CHKID LDA #$FF
0770  CMP HEADER
0780  BNE CONT
0800  CMP HEADER+1
0810  BEQ RDHDR ;ignore $FF,$FF id code
0820 ;
0830 ;Read end address
0840 ;
0850 CONT LDA #HEADER+2&255
0860  STA ICBAL,X
0870  LDA #HEADER+2/256
0880  STA ICBAL+1,X
0890  JSR CIO ;get end address
0900  BPL HDROK
0930  BMI ERROR ;bad file
0940 ;
0950 ;Store program start address
0960 ;
0970 HDROK LDA FLAG
0980  BEQ RDBLOK ;skip if not 1st block
0990  LDA HEADER ;use start of program
```

```
1000  STA RUNLOC ;  as default run adr
1010  LDA HEADER+1
1020  STA RUNLOC+1
1030  LDA #0
1040  STA FLAG ;clear 1st block flag
1050 ;
1060 ;Read a block of object code
1070 ;
1080 RDBLOK LDA HEADER ;load address
1090  STA ICBAL,X
1100  LDA HEADER+1
1110  STA ICBAL+1,X
1120  LDA HEADER+2 ;end address
1130  SEC
1140  SBC HEADER ;length = end - start
1150  STA ICBLL,X
1160  LDA HEADER+3
1170  SBC HEADER+1
1180  STA ICBLL+1,X
1190  INC ICBLL,X ;adjust length by 1
1200  BNE *+5
1210  INC ICBLL+1,X
1220 ;
1230  JSR CIO ;read block
1240  BPL RDHDR ;get next block
1250  CPY #ENR
1260  BEQ RDHDR ;this is also OK
1270  JMP ERROR ;bad file
1280 ;
1290 ;Subroutines follow
1300 ;
1310 ;***********************
1320 ;Start selected program
1330 ;***********************
1340 DONE JSR CLOSEIT
1350  JMP (RUNLOC) ;start program
1360 ;****************************
1370 ;Return error code to BASIC
1380 ;****************************
1390 ERROR TYA
1400  STA 212 ;tell BASIC what's wrong
1410  LDA #0
1420  STA 213
1430 ;now fall through to CLOSEIT
1440 ;then return to BASIC
1450 ;**************
1460 ;Close the IOCB
1470 ;**************
1480 CLOSEIT LDA #CLOSE
```

```
1490   STA ICCOM,X
1500   JSR CIO ;close file
1510   RTS
```

**PROGRAM 2. Loading Binary DOS Files From BASIC.**

```
10 REM LOADIT demo menu
20 REM by Robert E. Alleger
30 DIM LINE$(15),DIR$(12*64),DRIVE$(3)
40 REM * Initialization *
50 GRAPHICS 0:POKE 752,1
60 DRIVE$="D1:"
70 PRINT ,"[illegible] for Drive ";DRIVE$:PRINT
80 GOSUB 5000:REM store LOADIT.OBJ
90 LINE$=DRIVE$:LINE$(4)="*.*"
100 DIR$(1,1)=" ":DIR$(12*64)=" "
110 DIR$(2)=DIR$
120 CLOSE #1:OPEN #1,6,0,LINE$
130 TRAP 380:ENTRY=1:LINEFLAG=1
140 REM * Read the directory *
150 FOR FILENUMBER=1 TO 64
160 INPUT #1,LINE$
170 IF LINE$(2,2)<>" " THEN 380
180 PD=ENTRY
190 REM * Scan file name *
200 FOR PS=3 TO 10
210 IF LINE$(PS,PS)=" " THEN 240
220 DIR$(PD,PD)=LINE$(PS,PS)
230 PD=PD+1:NEXT PS
240 REM * Check for extension *
250 IF LINE$(11,11)=" " THEN 320:REM no exte
    nsion
260 DIR$(PD,PD)=".":REM append dot
270 PD=PD+1
280 REM * Scan file extension *
290 FOR PS=11 TO 13
300 DIR$(PD,PD)=LINE$(PS,PS)
310 PD=PD+1:NEXT PS
320 REM * Display file name.ext *
330 IF LINEFLAG=3 THEN PRINT :LINEFLAG=1
340 IF FILENUMBER<10 THEN PRINT " ";
350 PRINT FILENUMBER;" ";DIR$(ENTRY,ENTRY+11
    );"  ";
360 LINEFLAG=LINEFLAG+1
370 ENTRY=ENTRY+12:NEXT FILENUMBER
380 REM * Choose one *
390 PRINT :PRINT "Enter number of file to lo
    ad: ";
400 TRAP 390:INPUT N
410 IF N<1 OR N>FILENUMBER-1 THEN 390
420 LINE$=DRIVE$
430 LINE$(4)=DIR$(N*12-11,N*12)
440 GRAPHICS 0:POSITION 2,10
450 PRINT "{6 SPACES}LOADING  ";LINE$
```

```
460 REM * See if machine language *
470 FOR PS=4 TO 12
480 IF LINE$(PS,PS+3)=".CMD" THEN 540
490 NEXT PS
500 REM * Load BASIC program *
510 TRAP 530
520 RUN LINE$
530 ERROR=PEEK(195):GOTO 560
540 REM * Load M.L. program *
550 ERROR=USR(1536,ADR(LINE$))
560 REM * Shouldn't be here! *
570 PRINT "ERROR #";ERROR;" encountered duri
    ng load"
580 END
5000 FOR A=1536 TO 1717:READ B:POKE A,B:NEXT
      A
5001 DATA 162,16,32,173,6,134,207,104,104,15
     7,69,3,104,157,68,3,169,4,157,74,3,169,
     3,157,66
5002 DATA 3,32,86,228,16,3,76,166,6,169,203,
     157,68,3,169,0,157,69,3,169,2,157,72,3,
     169
5003 DATA 0,157,73,3,169,7,157,66,3,32,86,22
     8,16,6,192,136,240,92,208,96,169,255,19
     7,203,208
5004 DATA 4,197,204,240,210,169,205,157,68,3
     ,169,0,157,69,3,32,86,228,16,2,48,69,16
     5,207,240
5005 DATA 14,165,203,141,224,2,165,204,141,2
     25,2,169,0,133,207,165,203,157,68,3,165
     ,204,157,69,3
5006 DATA 165,205,56,229,203,157,72,3,165,20
     6,229,204,157,73,3,254,72,3,208,3,254,7
     3,3,32,86
5007 DATA 228,16,137,192,3,240,133,76,166,6,
     32,173,6,108,224,2,152,133,212,169,0,13
     3,213,169,12
5008 DATA 157,66,3,32,86,228,96
5009 RETURN
```

**PROGRAM 3. Loading Binary DOS Files From BASIC.**

```
10 REM DATAGEN
20 REM Translates DOS LOAD files
30 REM to BASIC DATA statements
40 REM (C) 1981 by Robert E. Alleger
50 REM * Initialization *
60 DIM FI$(15),FO$(15)
70 ERRSAV=195:MAX=25
80 REM * ask for file specs *
90 GRAPHICS 0
100 PRINT ,"DATAGEN":PRINT
110 TRAP 120
120 PRINT "Input file spec:  ";
130 INPUT FI$
140 TRAP 150
150 PRINT "Output file spec: ";
160 INPUT FO$
170 TRAP 610:CLOSE #1:CLOSE #2
180 OPEN #1,4,0,FI$
190 OPEN #2,8,0,FO$
200 REM * Get header ID (2 bytes) *
210 GET #1,B1:GET #1,B2
220 IF (B1=132 AND B2=152) OR (B1=255 AND B2
    =255) THEN 260
230 PRINT "{BELL}Not LOAD format":GOTO 620
240 REM * Ask for starting line *
250 TRAP 260
260 PRINT "Starting line number: ";
270 INPUT LNBR
280 REM * Get START & END addresses *
290 TRAP 560:REM trap normal EOF
300 GET #1,B1:GET #1,B2
310 ADRSTART=B1+B2*256
320 IF ADRSTART=65535 THEN 300
330 GET #1,B1:GET #1,B2
340 ADREND=B1+B2*256
350 TRAP 540:REM trap premature EOF
360 REM * Build FOR/NEXT loop *
370 PRINT #2;STR$(LNBR);
380 PRINT #2;"FOR A=";STR$(ADRSTART);" TO ";
    STR$(ADREND);
390 PRINT #2;":READ B:POKE A,B:NEXT A"
400 A=ADRSTART-1
410 REM * Build DATA statements *
420 LNBR=LNBR+1
430 IF A+1>ADREND THEN 520
440 IF LNBR>32765 THEN PRINT "{BELL}Line num
    ber too large":GOTO 620
450 PRINT #2;STR$(LNBR);"DATA ";
```

```
460 FOR N=1 TO MAX:A=A+1
470 IF A>ADREND THEN 520
480 IF N>1 THEN PRINT #2;",";
490 GET #1,B1:PRINT #2;STR$(B1);
500 IF N=MAX THEN PRINT #2
510 NEXT N:GOTO 410
520 PRINT #2:LNBR=LNBR+1
530 GOTO 290
540 IF PEEK(ERRSAV)=136 THEN PRINT "Prematur
    e EOF on input file":GOTO 620
550 GOTO 610
560 REM * Error TRAP *
570 IF PEEK(ERRSAV)<>136 THEN 610
580 PRINT #2;STR$(LNBR);"RETURN"
590 PRINT "<<< DONE >>>"
600 GOTO 620
610 PRINT :PRINT "{BELL}ERROR #";PEEK(ERRSAV
    )
620 CLOSE #1:CLOSE #2:TRAP 65535
```

**PROGRAM 4. Loading Binary DOS Files From BASIC.**

```
10 REM DSPLOAD
20 REM Display DOS LOAD
30 REM format information
40 REM (C) 1981 by Robert E. Alleger
50 REM * Initialization *
60 DIM F$(15),HEX$(4)
70 GRAPHICS 0
80 PRINT ,"DSPLOAD":PRINT :PRINT
90 PRINT "This program will print informatio
   n"
100 PRINT "for DOS LOAD format files."
110 TRAP 110:CLOSE #1
120 PRINT :PRINT "File spec: ";
130 INPUT F$
140 OPEN #1,4,0,F$
150 PRINT "{CLEAR}","LOAD Display":PRINT :PR
    INT
160 PRINT "File name",F$
170 PRINT "Format",,
180 REM * GET header ID *
190 TRAP 530
200 GET #1,B1:GET #1,B2
210 IF B1=132 AND B2=9 THEN PRINT "DOS 1 LOA
    D":GOTO 240
220 IF B1=255 AND B2=255 THEN PRINT "DOS 2 L
    OAD":GOTO 240
230 PRINT "{BELL}Not LOAD format{DOWN}":GOTO
     730
240 REM * Get START & END Addresses *
250 TRAP 590:GET #1,B1
260 TRAP 530:GET #1,B2
270 NBR=B1+B2*256
280 IF NBR=65535 THEN 310
290 ADRSTART=NBR:GOSUB 460
300 PRINT "Start - End Address ";HEX$;" - ";
310 GET #1,B1:GET #1,B2
320 NBR=B1+B2*256
330 ADREND=NBR:GOSUB 460
340 PRINT HEX$
350 IF ADREND<ADRSTART THEN 710
360 REM * Read LOAD file *
370 TRAP 560
380 FOR N=ADRSTART TO ADREND
390 GET #1,B1:BYTES=BYTES+1
400 IF N=736 THEN ADRAUTOL=B1
410 IF N=737 THEN ADRAUTOH=B1
420 IF N=738 THEN ADRINITL=B1
430 IF N=739 THEN ADRINITH=B1
```

```
440 NEXT N
450 GOTO 240:REM get next LOAD block
460 REM * Convert decimal to hex *
470 I=4:HEX$="0000"
480 T=NBR:NBR=INT(NBR/16):T=T-NBR*16
490 IF T<10 THEN HEX$(I,I)=STR$(T):GOTO 510
500 HEX$(I,I)=CHR$(T+55)
510 IF NBR<>0 THEN I=I-1:GOTO 480
520 RETURN
530 REM * ERROR #1 *
540 PRINT :PRINT "{BELL}Premature EOF while
    reading HEADER"
550 GOTO 730
560 REM * ERROR #2 *
570 PRINT :PRINT "{BELL}Premature EOF while
    reading DATA"
580 GOTO 730
590 REM * ERROR #3 *
600 IF ADRAUTOL=0 AND ADRAUTOH=0 THEN 640
610 NBR=ADRAUTOL+ADRAUTOH*256
620 GOSUB 460
630 PRINT "Auto Run Address",HEX$
640 IF ADRINITL=0 AND ADRINITH=0 THEN 680
650 NBR=ADRINITL+ADRINITH*256
660 GOSUB 460
670 PRINT "Init Address",HEX$
680 PRINT "Program size",INT(BYTES/1024*100)
    /100;"K BYTES"
690 PRINT :PRINT "{BELL}---EOF---"
700 GOTO 730
710 REM * ERROR #4 *
720 PRINT :PRINT "{BELL}END Address less tha
    n START Address"
730 REM * Exit *
740 TRAP 65535:CLOSE #1:END
```

# The Resident Disk Handler

Frank Kastenholz

*This technical article explores, with commentary and examples, the use of the operating system's Resident Disk Handler for accessing disk sectors . . . without DOS. If you're interested in learning more about the Atari DOS itself, see* Inside Atari DOS *from* **COMPUTE! Books**.

Would you like to be able to hide data on your disks without having DOS signal its presence with a file name and directory entry? Would you like to be able to access *any* sector on a disk, independent of DOS? You could check the disks that you are using for bad sectors without destroying what's on the disk. You could create your own unique disk format to suit your own unique needs.

The keys that open the door to this wonderland of *direct access storage* are the Resident Disk Handler and the Device Control Block.

The Resident Disk Handler is a section of code that exists in the ROMs of your Atari computer (both the 400 and the 800), and the Device Control Block is a section of RAM that contains the various parameters which control the actions of the Resident Disk Handler. The Resident Disk Handler is capable of performing four different operations: Get Sector, Put Sector with Verify, Status Request, and Format Disk. For the purposes of disk access only the first two operations are important, and I shall discuss only those two operations.

The Get Sector operation will retrieve any one sector from the disk and place it in any 128 byte block of RAM. The Put Sector with Verify operation will take 128 bytes of data anywhere from memory (RAM or ROM) and write that data to any sector on the disk. It will then check to make sure that the data were written correctly.

And how, I hear you cry, does one make use of this miracle of modern science? To use the Resident Disk Handler is an extremely simple task. It just sounds hard. At the end of this article I have included a program that should adequately demonstrate the basics of using the Resident Disk Handler.

The key to using the Resident Disk Handler is a chunk of memory called the Device Control Block. The Device Control Block is similar in function to the IOCBs that are used in BASIC. The Device Control Block is 11 bytes long and begins at location $0300 (768 decimal). (All hexadecimal numbers are preceded by a $ and followed by their decimal equivalent in parentheses.) For our purposes only seven bytes are needed, the other four being used internally by the Resident Disk Handler. The bytes we shall use are the device unit number byte, the command number byte, the status byte, the two buffer address bytes, and the two sector number bytes.

The device unit number byte is located at location $0301 (769) and contains the unit number of the disk drive you wish to access (1, 2, 3, or 4).

The command number byte is at location $0302 (770) and contains the command number of the operation to be performed. The command number for the Get Sector operation is $52 (82), and the number for the Put Sector with Verify operation is $57 (87).

The status byte is the only byte that you do not have to put something into before you use the Resident Disk Handler. It sets up the status byte to reflect the success (or lack of it) of the operation that was just attempted. The status byte is at location $0303 (771) and may have one of seven values. A 1 in this byte indicates that the operation was completed successfully. The other six values indicate an error occurred during the operation. The values are $8A (138), $8B (139), $8C (140), $8E (142), $8F (143), and $90 (144). The meaning of these status codes is the same as for the error codes of the same number in BASIC.

The two buffer address bytes are at locations $0304 (772) for the low order byte of the address and location $0305 (773) for the high order byte of the address. These two bytes contain the address in memory of the source of the data, for a Put Sector with Verify, or the destination of the data, for a Get Sector. To set these bytes up in BASIC you must divide the address that is to be used by 256 and place the remainder in byte 772 and the quotient in byte 773.

The two sector number bytes contain the number of the sector on the disk that is to be accessed. This number may be any number from 1 to 720 inclusive. If you were not the trusting type you might say "Sector 720 is not addressable – it says so in the DOS manual." But since you are the trusting type, you will not say that, and I will not be forced to reply "True, but we are not using DOS here and sector 720 *is* addressable when using the Resident Disk Handler!"

Once the Device Control Block has been properly set up you have to call the Resident Disk Handler so it can do its work. If you are programming in machine language, this is a trivial job. You merely do a JSR $E453. It will do its work and then do a RTS when it is done. Nothing could be simpler. If you are a BASIC programmer it is slightly harder to call the Resident Disk Handler. You must load the following assembly code into RAM and then do a USR to it.

**PROGRAM 1.**

| Object Code HEX | Decimal | Source Code | Comments |
|---|---|---|---|
| $68 | 104 | PLA | The Extra PLA required by USR. |
| $20 $53 $E4 | 32 83 228 | JSR $E453 | Call the Resident Disk Handler. |
| $60 | 96 | RTS | Return to BASIC. |

The BASIC code in Program 2 will load the assembler code of Program 1 into RAM beginning at location 1536. To call the Resident Disk Handler you would then do a X=USR(1536).

**PROGRAM 2.**

```
10 DATA 104,32,83,228,96
20 FOR I=1536 TO 1540
30 READ J:POKE I,J
40 NEXT I
```

The code in Program 3 is a short BASIC program that will show you how to use the Resident Disk Handler. The program will either put or get one sector of data. If you get a sector of data, the program will print out that data as character data (if a byte is a 65, it will print A). If you are going to put a sector of data to the disk, the program will ask you to enter the data to put in character form. Comments on the program follow the listing.

**PROGRAM 3.**

```
10 DATA 104,32,83,228,96
20 FOR I=1536 TO 1540
30 READ J:POKE I,J
40 NEXT I
50 DIM A$(128)
60 PRINT "GET OR PUT";:INPUT A$
70 IF A$="GET" THEN 100
80 IF A$="PUT" THEN 200
90 GOTO 60
100 LET COMMAND=82
110 GOSUB 1000
120 STAT=PEEK(771)
130 IF STAT=1 THEN 160
140 PRINT "ERROR #";STAT;" ON GET"
```

```
150 GOTO 60
160 FOR I=0 TO 127
170 A$(I+1,I+1)=CHR$(PEEK(1664+I)):NEXT I
180 PRINT A$
190 GOTO 60
200 FOR I=1 TO 128:A$(I,I)=" ":NEXT I
210 PRINT "ENTER DATA";:INPUT A$
220 FOR I=0 TO LEN(A$)-1
230 POKE 1664+1,ASC(A$(I+1,I+1)):NEXT I
240 FOR I=LEN(A$) TO 127
250 POKE 1664+I,0:NEXT I:LET COMMAND=87
260 GOSUB 1000
270 STAT=PEEK(771)
280 IF STAT=1 THEN PRINT "OPERATION COMPLETE
    ":GOTO 60
290 PRINT "ERROR #";STAT;" ON PUT":GOTO 60
1000 REM DISK ACCESS ROUTINE FOLLOWS
1010 PRINT "SECTOR NUMBER TO ACCESS";:INPUT
     SNUM
1020 POKE 779,INT(SNUM/256):POKE 778,INT((SN
     UM/256-INT(SNUM/256))*256)
1030 POKE 769,1
1040 POKE 772,128:POKE 773,6
1050 POKE 770,COMMAND
1060 X=USR(1536)
1070 RETURN
```

**Lines 10-40** load the short assembler routine needed to call the Resident Disk Handler into memory.

**Line 50** dimensions A$ as a text string so we can use it to store data.

**Lines 60-90** input an operation from the keyboard, determine which operation it is, and jump to the appropriate routine to handle that operation.

**Lines 100-190** are the Get operation.

**Line 100** sets the command as Get Sector.

**Line 110** calls the disk access subroutine.

**Line 120** sets STAT equal to the status of the operation.

**Line 130** determines if the operation was successful. If it was it goes to **Line 160.**

**Lines 140 and 150** print an error message and then start another operation.

**Lines 160, 170, and 180** get the input data from the input buffer, put them into A$ and print them.

**Line 190** starts another operation.

**Lines 200-290** are the Put Sector Lines.

**Line 200** clears A$ out.

**Line 210** enters the data from the keyboard.

**Lines 220 and 230** put the output data into the output buffer.

**Lines 240 and 250** fill any bytes remaining after the last data byte and the 128th byte of the buffer with zeros. Line 240 also sets the command to the Put Sector with Verify operation.

**Line 260** calls the disk access subroutine.

**Line 270** sets STAT equal to the status of the operation.

**Line 280** determines if the operation was successful or not and if it was, prints "OPERATION COMPLETE" and starts another operation.

**Line 290** prints an error message and then starts another operation.

**Lines 1000-1070** set up the Device Control Block and then call the Resident Disk Handler.

**Line 1010** inputs the sector number to access.

**Line 1020** puts the sector number into the sector number bytes. The first POKE statement takes the quotient of the sector number divided by 256 and puts it into the high byte of the two sector number bytes. The second POKE statement takes the remainder of the sector number divided by 256 and puts it into the low byte of the sector number bytes.

**Line 1030** sets the unit number to 1.

**Line 1040** sets the two buffer address bytes to 1664.

**Line 1050** sets up the Command Number Byte.

**Line 1060** calls the short assembler routine which then calls the Resident Disk Handler.

**Line 1070** returns to the main routine.

This program is tutorial, intended to help you understand the Resident Disk Handler, what it is and how to use it. If you wish to use the Resident Disk Handler, you would want to make some improvements to the subroutine that I have presented here. First you would pass the sector number from the main program instead of entering it from the keyboard. Other improvements would include combining some of the statements to make the program shorter, or converting the program into machine language. One improvement that I have found particularly valuable is to have the subroutine repeat an error several times if an error occurs. Since most disk errors are recoverable if the operation is retried, this has the effect of reducing the number of disk errors I get to almost nothing.

Now that you have finished reading this article, I can hear you grumbling "This just looks like a hard way of doing a POINT!" But it isn't! The Resident Disk Handler allows you to access any sector on the disk. The POINT command in BASIC allows you to access any sector in a particular file. If you POINT to sector 3, you will access sector 3 of the file, and not absolute sector 3. The Resident Disk Handler will get you absolute sector 3. It can access a sector that is un-allocated, POINT can't. It can access the sectors that DOS uses, POINT can't.

While using these methods takes more work than just OPENing, PRINTing, and CLOSEing a file in BASIC, the added flexibility more than compensates for the extra work.

# LISTING CONVENTIONS

In order to make special characters, inverse video, and cursor characters easy to type in, **COMPUTE!** magazine's new Atari listing conventions are used in all the program listings in this book.

Please refer to the following tables and explanations if you come across an unusual symbol in a program listing.

Characters in inverse video will appear like: INVERSE VIDEO
Enter these characters with the Atari logo key, " ⅄ ".

Graphics characters, such as CTRL-T, the ball character ● will appear as the "normal" letter enclosed in braces, e.g. {T}.

A series of identical control characters, such as 10 spaces, three cursor-lefts, or 20 CTRL-R's, will appear as {10 SPACES}, {3 LEFT}, {20 R}, etc. If the character in braces is in inverse video, that character or characters should be entered with the Atari logo key. For example, ♥ means to enter a reverse-field heart with CTRL-comma, {5 U} means to enter five inverse-video CTRL-U's.

# INDEX

ISBN 0-942386-06-X