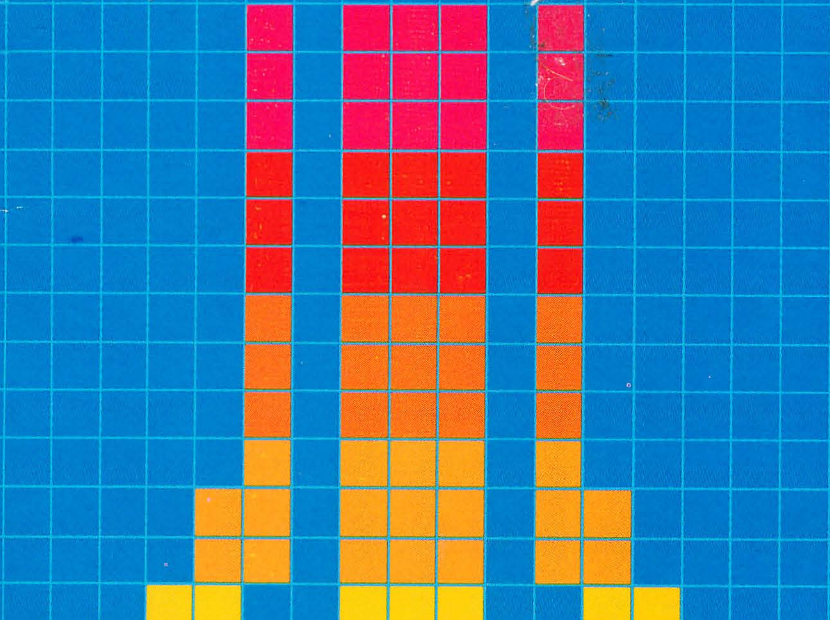
COMPUTE!'s SECOND BOOK OF ATARI GRAPHICS



Graphics, utilities, tutorials, programs and other helpful information for the users of the Atari® personal computers.

A **COMPUTE! Books** Publication

$12.95

COMPUTE!'s SECOND BOOK OF ATARI GRAPHICS

COMPUTE!™ Publications, Inc. abc
One of the ABC Publishing Companies

Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-28-0

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a subsidiary of American Broadcasting Companies, Inc., and is not associated with any manufacturer of personal computers. Atari is a trademark of Atarı, Inc.

Contents

Foreword

Whether you are a beginner to Atari Graphics or have been programming graphics for years, *COMPUTE!'s Second Book of Atari Graphics* will prove to be an invaluable resource. There are articles for beginners and utilities that both novices and professionals will find useful.

As with all COMPUTE! publications, you'll find a wide range of easy-to-understand tutorials, and high quality programs collected for your use and enjoyment.

If you already have *COMPUTE!'s First Book of Atari Graphics,* you know how useful it is; we know you will find this book just as valuable. And if this is your first COMPUTE! publication, you are in for some pleasant surprises.

1

Fundamentals

1

Graphics in Mode 0

Stephen Levy

The Atari character set makes available numerous graphic characters. Some ways to exploit this feature are explored here.

Atari, long known for its superior graphics, has one graphics mode which is rarely used for graphics. Graphics mode 0 is generally used only for text and text-oriented programs. Well, why not use graphics in mode 0?

Plenty to Choose from

Unlike modes 3 through 11, graphics mode 0 has its own set of characters built-in which can be used to create some very nice designs. The short program below will show most of the graphic characters that are available in mode 0.

```
10 POKE 82,9:POKE 83,25
20 GRAPHICS 0:CHAR=0:LINE=0:PRINT
30 PRINT CHR$(CHAR);CHR$(32);
40 CHAR=CHAR+1:LINE=LINE+1
50 IF LINE=8 THEN LINE=0:PRINT :PRIN
   T
60 IF (CHAR>26 AND CHAR<32) OR (CHAR
   >155) THEN PRINT CHR$(27);
70 IF CHAR=32 THEN CHAR=128:PRINT
80 IF CHAR=155 THEN GOTO 40
90 IF CHAR=160 THEN PRINT :GOTO 110
100 GOTO 30
110 POKE 82,2:POKE 83,39:END
```

What is seen on the screen when this program is RUN are the ATASCII characters from 0 to 31 and 128 to 159. (A complete list of the character set can be found in Appendix C of the *Atari 400/800 BASIC Reference Manual.)* Characters 128 to 159 are the inverse characters.

Centering the Display

The display has been centered for two reasons: it looks nicer that way, and it illustrates a way to place characters on the screen other than at the left margin.

Actually, the first character on each line is at the left margin. Look at line 10. Memory location 82 stores information the computer needs to find the correct column for the left margin. When you turn on the computer, a 2 is stored in this location. Likewise, memory location 83 stores the column of the right margin. All that has been done here is to change the margins to 9 and 25. On line 110 the margins are restored to 2 and 39.

There are a number of other ways to place characters on the screen at a specific place, including the POSITION, PLOT, and DRAWTO statements. More information about these statements will be provided a little later.

Printing the Graphic Characters

There are two ways to print each character in the character set. Printing the letter A can be done with either

```
    PRINT "A"
       or
  PRINT CHR$(65)
```

Both methods produce the same results. The second method can be very inefficient for printing text to the screen, but it's very efficient for printing characters with consecutive ATASCII numbers and graphic characters.

As an example, enter this line and press RETURN:

```
FOR I=65 TO 90:PRINT CHR$(I),:NEXT I
```

Special Characters

Characters 27 to 31, 125 to 127, 156 to 159, and 253 to 255 need special instructions in order to be printed. Note that line 60 checks for some of those characters. When it finds one, it prints character 27 (the ESCape character) before printing the character.

An alternative method can be used to print these characters. Make these changes in the program: delete line 60, add line 15,

```
15 POKE 766,2
```

and change line 110 to read

```
110 POKE 82,2:POKE 83,39:POKE 766,0:END
```

Try typing in the following program. It can be difficult to enter correctly if you are not experienced at entering these characters. When the program is RUN, the results should be four short lines printed on the screen. The first line should have the ESCape symbol. The second and third lines should show backward curved arrows (one will be inverse video); the last two lines should be the delete symbol and four arrows.

If you do not get the correct results, the proper keystrokes are given after the listing.

```
10 PRINT "{2 ESC}"
20 PRINT "{ESC}{CLEAR}"
30 PRINT "{ESC}{BELL}"
40 PRINT "{ESC}{DELETE}"
50 PRINT "{ESC}{UP}{ESC}{DOWN}{ESC}
   {LEFT}{ESC}{RIGHT}"
```

To enter this program, type each line number, the word PRINT, a quote symbol ("), and the following keys:

ESC, ESC, ESC, ESC, RETURN for line 10;
ESC, ESC, ESC, CTRL-CLEAR, RETURN for line 20;
ESC, ESC, ESC, CTRL-2, RETURN for line 30;
ESC, ESC, ESC, CTRL-DELETE/BACK S, RETURN for line 40;
ESC, ESC, ESC, CTRL-UP arrow, ESC, ESC, ESC, CTRL-DOWN arrow, ESC, ESC, ESC, CTRL-LEFT arrow, ESC, ESC, ESC, CTRL-RIGHT arrow, RETURN for line 50.

The Easy Way

The program below prints the same characters as the previous program. By POKEing a 2 into memory location 766, you no longer need to press the ESC key three times for each character, but just once.

```
5 POKE 766,2
10 PRINT "{ESC}"
20 PRINT "{CLEAR}"
30 PRINT "{BELL}"
40 PRINT "{DELETE}"
50 PRINT "{UP}{DOWN}{LEFT}{RIGHT}"
60 POKE 766,0
```

PLOT and DRAWTO

Any character can be used with the PLOT and DRAWTO command. The following program is self-explanatory and should give some idea of how to use these two commands in graphics mode 0.

```
10 GRAPHICS 0:POKE 752,1:REM SET GRA
   PHICS MODE AND TURN OFF CURSOR
20 PRINT "{DOWN}":COLOR 148:REM ATAS
   CII FOR CHARACTER TO PLOT
```

```
30 PLOT 2,4:DRAWTO 36,4
40 DRAWTO 36,18:DRAWTO 2,18
50 DRAWTO 2,3:REM NOTICE WE DRAW ONE
    POSITION PAST OUR STARTING POINT
60 POSITION 5,7:PRINT "PRINT YOUR ME
   SSAGE HERE. BE"
70 POSITION 5,9:PRINT "SURE NOT TO P
   UT TOO MANY"
80 POSITION 5,11:PRINT "CHARACTERS O
   N A LINE."
90 POSITION 2,20:REM MOVE CURSOR OUT
    OF THE WAY
100 POKE 752,0:REM TURN CURSOR BACK
    ON
```

POSITIONing Characters

Any character can be placed on the screen at any location chosen. The POSITION statement is easy to use if you follow certain rules:

• The format for POSITION is POSITION *c,r* where *c* is the desired column (0 to 39), and *r* is the desired row (0 to 23).

• A new POSITION statement will cause the next PRINT command to print over any characters which are at that location.

• POSITION places the cursor anywhere on the screen.

• Be careful printing to the thirty-ninth position on line — it will cause a line to wrap around to the next line. Likewise, be careful using the twenty-third line—it can cause the screen to scroll and lose the top line.

• It is a good idea to first erase the spaces to be used.

The next program also uses the POSITION statement, but this time a loop is used to change the locations.

```
100 DIM TRIANGLE$(34)
110 GRAPHICS 0:POKE 752,1:SETCOLOR 2
    ,7,14:SETCOLOR 1,7,4
120 C=18:R=1
130 TRIANGLE$="{H}{J}"
140 POSITION C,R:PRINT TRIANGLE$
150 FOR B=2 TO 32 STEP 2
160 FOR A=2 TO B
170 TRIANGLE$(A,A)="■"
180 NEXT A
190 TRIANGLE$(A,A+1)="■{J}"
200 C=C-1:R=R+1
210 POSITION C,R:PRINT TRIANGLE$
220 NEXT B
230 POKE 752,0
```

Now add these few lines to the previous program and watch what happens to the triangle:

```
25Ø FOR C=1 TO 25
26Ø A=1:B=2:GOSUB 3ØØ
27Ø A=2:B=1:GOSUB 3ØØ
28Ø NEXT C
29Ø END
3ØØ SETCOLOR B,7,4:SETCOLOR A,7,14
31Ø FOR D=1 TO 5Ø:NEXT D:RETURN
```

Drawing a Picture

The next two listings draw a simple picture. Both listings do exactly the same thing. Both versions of the drawing are included here as an illustration of two methods to print to the screen. When you're creating the picture, the first method is often easier because the picture can be seen while it is LISTed. The second method is supplied for purposes of comparison.

Program 1. Drawing with Characters

```
1Ø GRAPHICS Ø:SETCOLOR 4,3,6:SETCOLOR
    2,11,14:SETCOLOR 1,11,2:POKE 752,
    1
15 POKE 559,Ø:REM THIS LINE OPTIONAL
2Ø POSITION 14,6:PRINT " {H}{J}
    {3 SPACES}{H}{J}"
3Ø POSITION 14,7:PRINT "{B}{7 M}{V}"
4Ø POSITION 14,8:PRINT "{B}{Q}{W}{E}
    {Q}{W}{E}{V}"
5Ø POSITION 14,9:PRINT "{B}{A}{S}{D}
    {A}{S}{D}{V}"
6Ø POSITION 14,1Ø:PRINT "{B}{Z}{X}
    {C} {Z}{X}{C}{V}"
7Ø POSITION 14,11:PRINT "{B}
    {4 SPACES}{H}  {V}"
8Ø POSITION 14,12:PRINT "{B} {B}{2 M}
    ■  {V}"
9Ø POSITION 14,13:PRINT "{B} {B}   ■
    {V}"
1ØØ POSITION 14,14:PRINT "{B} {B}   ■
     {V}"
11Ø POSITION 14,15:PRINT "{B} {B}{2 N}
    ■  {V}"
12Ø POSITION 14,16:PRINT "{B}{7 N}
    {V}"
13Ø PRINT "{3 DOWN} A PICTURE IS WORT
    H A THOUSAND WORDS"
14Ø POKE 559,34
15Ø GOTO 15Ø
```

Program 2. Drawing with CHR$

```
10 GRAPHICS 0:SETCOLOR 4,3,6:SETCOLOR
    2,11,14:SETCOLOR 1,11,2:POKE 752,
   1
15 POKE 559,0:REM THIS LINE OPTIONAL
20 FOR I=6 TO 16
30 POSITION 14,I
40 FOR H=1 TO 9
50 READ A
60 PRINT CHR$(A);
70 NEXT H
80 NEXT I
90 PRINT :PRINT "{3 DOWN} A PICTURE I
   S WORTH A THOUSAND WORDS"
100 POKE 559,34
110 GOTO 110
200 DATA 32,8,10,32,32,32,8,10,32
210 DATA 2,13,13,13,13,13,13,13,22
220 DATA 2,17,23,5,32,17,23,5,22
230 DATA 2,1,19,4,32,1,19,4,22
240 DATA 2,26,24,3,32,26,24,3,22
250 DATA 2,32,32,32,32,8,32,32,22
260 DATA 2,32,2,13,13,160,32,32,22
270 DATA 2,32,2,32,32,160,32,32,22
280 DATA 2,32,2,32,32,160,32,32,22
290 DATA 2,32,2,14,14,160,32,32,22
300 DATA 2,14,14,14,14,14,14,14,22
```

Discovering "Hidden" Graphics

Gregory L. Kopp

GRAPHICS 1 and 2, the large-text modes, split the normal character set in two and allow only half to be used at once. Both modes also handle colors differently. Here's how to take advantage of those quirks.

If you were a stumbling, beginning BASIC programmer like I was, you probably tried to enter a few "improper" graphics commands which resulted in curious and unexpected displays on your television screen. Before I understood the function and proper use of POKE 756 (which displays lowercase letters and special graphics characters in text modes 1 and 2), I stubbornly tried to put control characters onscreen *without* the requisite POKE, which produced only seemingly random keyboard characters and frustration instead.

Much later, the thought nevertheless occurred to me that I might have accidentally discovered some "hidden" (or at least undocumented) graphics capability of my Atari. In the experimental binge to which owners of microcomputers are sometimes given, I used the PRINT #6; command to enter each keyboard character while pressing CTRL at the same time. Discovery! Although the Atari special graphics characters appeared in the PRINT #6; statement, the actual screen display consisted of *keyboard* characters, but *not* the characters for the keys I entered.

Dutifully noting the results (Tables 1 and 2), I pondered the apparent micro-fluke, these "hidden" characters, then asked myself the inevitable scientific question: "So what?" Two uses came fairly quickly to mind — the first purely cosmetic, the second functional.

If I could change these hidden characters from "default green" to other colors, I could eliminate the irksome problem encountered in modes 1 and 2 of having punctuation and numbers displayed in different colors from the text lettering. The INVERSE key! Sure enough, PRINTing the graphics characters in inverse changed my hidden green characters to red. Now I could choose from normal character (orange), inverse normal (blue), CTRL character (green) and inverse CTRL (red). Experimenting further, I discovered I could achieve *any* Atari color by use of a SETCOLOR 0 to 3 or POKE 708 to 711 command to change each respective character. No more would I have to sheepishly explain to those not-of-the-computer-persuasion why my apostrophe or my "1" was blue while my text was red!

So much for cosmetics. If you are not bothered by the inconsistent color text problem, then use the last two paragraphs as speed-reading exercises. However, if you have purchased software for redefining character sets (see Chapter 3), you may already have thought of the second application. Instead of redefining your lowercase character set (and thereby "losing" it) to achieve new characters, you may use "hidden graphics" to redefine the number set, selected punctuation marks, or arithmetic signs. While this could be done normally, using "hidden graphics" allows you to display numbers, punctuation, or signs in *four* colors instead of only two. (If you have not run Program 1 yet, try it. Then try to produce four different color 1's the conventional way.)

A Second Approach

Now enter and run Program 2.

If you are trying to figure out how we got all those alphabet characters using PLOT and COLOR statements, read on.

As any intermediate programmer can tell you, you cannot plot points in modes 1 and 2. You get absolutely nothing displayed if you try it. Of course, the stumbling beginner might think the reason you get nothing is that you did not enter a COLOR statement. Sandwiching COLOR 1 between the lines and trying again, you discover that you have plotted a ! instead of a point. "Pixel-head!" you chide yourself. "You can't use PLOT in modes 1 and 2!" You note this in your reference manual and rank yourself a step closer to intermediate programmer, missing the opportunity to discover more hidden graphics.

The Atari will plot a *character* in modes 1 and 2 at whatever position the programmer commands. The nature and color of that character are determined by a single COLOR statement. Using the COLOR Statement Graphics Chart (Table 3) you can display any Atari keyboard character (POKE 756, 226 for lowercase) by using the associated COLOR statement, then plotting X,Y coordinates to place it at the desired position on the screen.

Once again, SETCOLOR 0 to 3 or POKE 708 to 711 can be used to color each individual character, *including* lowercase characters which are normally limited to only two colors. (Note: These SETCOLORs and POKEs work only when using GR. 1 or 2 + *16*.) Again, redefined characters may be used and this time manipulated arithmetically. Game writers, rejoice!

While the PRINT #6; approach displays numbers, punctuation and arithmetic signs, the COLOR/PLOT technique allows access to upper- and lowercase letters as well. Preference for one method over the other will vary from user to user and application to application, as you will see once you have tried them a few times.

Table 1. Regular and Hidden Colors

Character	(Default)	SETCOLOR	POKE
Normal	orange	0	708
"Hidden"	green	1	709
Inverse, normal	blue	2	710
Inverse, "hidden"	red	3	711

Table 2. Hidden Graphics

To Get Character	Color	Press Keys
0	Green	CTRL P
	Red	INVERSE CTRL P
1	Green	CTRL Q
	Red	INVERSE CTRL Q
2	Green	CTRL R
	Red	INVERSE CTRL R
3	Green	CTRL S
	Red	INVERSE CTRL S
4	Green	CTRL T
	Red	INVERSE CTRL T
5	Green	CTRL U
	Red	INVERSE CTRL U
6	Green	CTRL V
	Red	INVERSE CTRL V
7	Green	CTRL W
	Red	INVERSE CTRL W
8	Green	CTRL X
	Red	INVERSE CTRL X
9	Green	CTRL Y
	Red	INVERSE CTRL Y
:	Green	CTRL Z
	Red	INVERSE CTRL Z
!	Green	CTRL A
	Red	INVERSE CTRL A
"	Green	CTRL B
	Red	INVERSE CTRL B
#	Green	CTRL C
	Red	INVERSE CTRL C
$	Green	CTRL D
	Red	INVERSE CTRL D
%	Green	CTRL E
	Red	INVERSE CTRL E
&	Green	CTRL F
	Red	INVERSE CTRL F
'	Green	CTRL G
	Red	INVERSE CTRL G
(	Green	CTRL H
	Red	INVERSE CTRL H
)	Green	CTRL I
	Red	INVERSE CTRL I
*	Green	CTRL J
	Red	INVERSE CTRL J

To Get Character	Color	Press Keys
+	Green	CTRL K
	Red	INVERSE CTRL K
,	Green	CTRL L
	Red	INVERSE CTRL L
–	Green	CTRL M
	Red	INVERSE CTRL M
.	Green	CTRL N
	Red	INVERSE CTRL N
/	Green	CTRL O
	Red	INVERSE CTRL O
[	Green	CTRL ;
	Red	INVERSE CTRL ;
@	Green	CTRL .
	Red	INVERSE CTRL .
∧	Green	ESC then BACK S
	Red	ESC then CTRL + DELETE
<	Green	ESC then CTRL + minus
	Red	ESC then SHIFT + DELETE
>	Green	ESC then CTRL + plus
	Red	ESC then CTRL + TAB
=	Green	ESC then CTRL + equals
	Red	ESC then SHIFT + INSERT
?	Green	ESC then CTRL + asterisk
	Red	ESC then SHIFT + TAB
—	Green	ESC then TAB
	Red	ESC then CTRL + INSERT
;	Green	ESC then ESC
]	Green	ESC then CTRL + CLEAR
	Red	ESC then CTRL + 2

*greens manipulated by SE.1 and POKE 709
reds manipulated by SE.3 and POKE 711

Table 3. COLOR Statements Graphics Chart

(SETCOLOR #)	0	1	2	3
(POKE register)	708	709	710	711
(default)	green	yellow	red	blue

COLOR Number*				Character
0	32	128	160	(space)
1	33	129	161	!
2	34	130	162	" (quotes)
3	35	131	163	#
4	36	132	164	$
5	37	133	165	%
6	38	134	166	&
7	39	135	167	' (apostrophe)
8	40	136	168	(
9	41	137	169	)
10	42	138	170	*
11	43	139	171	+
12	44	140	172	, (comma)
13	45	141	173	– (minus)
14	46	142	174	. (period)
15	47	143	175	/
16	48	144	176	0
17	49	145	177	1
18	50	146	178	2
19	51	147	179	3
20	52	148	180	4
21	53	149	181	5
22	54	150	182	6
23	55	151	183	7
24	56	152	184	8
25	57	153	185	9
26	58	154	186	:
27	59	†	187	;
28	60	156	188	⟨
29	61	157	189	=
30	62	158	190	⟩
31	63	159	191	?
96	64	224	192	@
97	65	225	193	A
98	66	226	194	B
99	67	227	195	C
100	68	228	196	D
101	69	229	197	E

102	70	230	198	F
103	71	231	199	G
104	72	232	200	H
105	73	233	201	I
106	74	234	202	J
107	75	235	203	K
108	76	236	204	L
109	77	237	205	M
110	78	238	206	N
111	79	239	207	O
112	80	240	208	P
113	81	241	209	Q
114	82	242	210	R
115	83	243	211	S
116	84	244	212	T
117	85	245	213	U
118	86	246	214	V
119	87	247	215	W
120	88	248	216	X
121	89	249	217	Y
122	90	250	218	Z
123	91	251	219	[
124	92	252	220	/
‡	93	253	221	]
126	94	254	222	∧
127	95	255	223	_ (underline)

*Use this statement format: Color *nn*: PLOT *x*, *y*. For example, to put a green A on the screen, use: COLOR 97: PLOT 0, 0

†COLOR 155: PLOT *x*, *y* puts a carriage return on the screen at the PLOT location. If you are also going to PRINT to the graphics screen using PRINT #6; "*n*", this command puts the PRINT #6 cursor in column 0 of the line *after* the line specified in PLOT *x*, *y*. In other words, if your program executes this line:

GRAPHICS 2:PRINT #6; "ABC"; :COLOR 155:PLOT 0,5:PRINT #6; "DEF" the letters ABC will be in line 0 and the letters DEF will be in line 6. To put a red semicolon on the screen, you must POKE the number 219 into the appropriate place in screen memory.

‡COLOR 125:PLOT *x*, *y* clears *both* the text window and the graphics screen. To put a green] on the screen you must POKE the number 125 into the appropriate place in screen memory.

Program 1. Hidden Graphics–1

```
10 GRAPHICS 2+16
20 X=0
30 FOR L=1 TO 50
40 RC=INT(15*RND(0)):RS=(255*RND(0))
50 SETCOLOR 0,RC,6
60 SOUND 0,RS,10,4
70 POSITION 5,4
80 ? #6;"1{Q}1{E}"
90 FOR W=1 TO 25:NEXT W
100 X=X+1:IF X=4 THEN X=0
110 NEXT L
120 SOUND 0,0,0,0
130 GRAPHICS 2+16
140 POSITION 5,4
150 ? #6;"1{Q}1{E}"
160 FOR W=1 TO 500:NEXT W
170 POSITION 2,7
180 ? #6;"HIDDEN GRAPHICS!"
190 GOTO 190
```

Program 2. Hidden Graphics–2

```
10 X=0:Y=0:Z=65
20 GRAPHICS 2+16
30 FOR AZ=0 TO 25
40 SOUND 0,255-AZ*10,AZ+8,8
50 COLOR Z
60 PLOT X,Y:IF X=18 THEN X=0:Y=Y+1
70 X=X+1:Z=Z+1
80 FOR W=1 TO 50:NEXT W
90 NEXT AZ
100 IF Z=91 THEN Z=193:X=0:Y=2:GOTO 30
110 IF Z=219 THEN Z=225:X=0:Y=4:GOTO 30
120 IF Z=219 THEN Z=225:X=0:Y=6:GOTO 30
130 FOR W=1 TO 500:NEXT W
140 POSITION 2,9:? #6;"COLOR STATEMEN
    T"
150 POSITION 4,10:? #6;"GRAPHICS!"
160 GOTO 160
```

1

Copy Your Screen to Your Printer

Harry A. Straw

This is a simple, short screen dump for mode 0. It will work with any BASIC program.

Here's a handy routine for copying text from your Atari screen (GRAPHICS 0 mode) to your printer. It is set up to use two GOSUB commands in your main program:

GOSUB 32010 to initialize
GOSUB 32040 each time you want to line-print a page displayed on your screen.

The program is straightforward, but a few comments may help you to run it smoothly.

The main business of this program is the double FOR/NEXT loop in lines 32050-32110. With the POSITION command, these loops move the cursor over the entire screen, one position at a time. At each cursor position, line 32080 GETs the ASCII number for the character under the cursor, and line 32090 puts the corresponding character on the printer. Since I have an 80-column printer and the Atari screen is only 40 characters wide, I need line 32105 to get printer carriage return at the proper place. You may be able to delete this line if you have a 40-column printer (or one that can be set to 40 columns).

Line 32040 (printer carriage return) makes sure that the printer head starts copying at its left-hand margin. Line 32120 "homes" the cursor at the end of the subroutine. This is not always necessary, but, depending on the next line in your main program, it may prevent an ERROR - 141, "cursor out of range."

You must OPEN a channel to GET from the screen. I use channel number 5, leaving channels 1-4 free for use in main programs. The initializing subroutine in lines 32010-32030 does this. It also expands the Atari display to its full 40-character width and 24-line height to match the cursor movement controlled by lines 32050 and 32060. The OPEN command clears the screen, so you must OPEN before displaying the text you want to copy. Just be sure your main program says GOSUB 32010 *ahead* of the screen display to be printed.

If you have only a few lines to copy, no problem. Merely adjust line 32050 to cover the rows you want to scan. Otherwise, the printer will run for all 24 rows, printing a lot of blank spaces wherever nothing shows on the screen.

There is no CLOSE #5 statement in the listing. This leaves channel 5 open, so it is not necessary to repeat GOSUB 32010 for each page to be line-printed.

Take advantage of Atari's ability to merge stored programs with RAM-resident programs by recording this routine with the command LIST"C:" and reading it with ENTER"C:" (or ENTER "D:Filename). CSAVE and CLOAD won't work this way. In fact, CLOAD erases programs in RAM! This routine starts with a high line number, 32000, so its line numbers won't conflict with those of a program already in RAM.

Sample Screen Dump

```
This is an example of a GRAPHICS 0
screen dumped to an Epson MX-80 printer
with the accompanying routine.


It is fast and easy to use, and will
work with almost any 80-column printer.

The routine can be added to your own
programs.

STOPPED  AT LINE 35

By typing GOTO 32040, we'll access the
routine in direct mode.  Use GOSUB
32040 in your programs.

GOSUB 32040
```

GRAPHICS 0 Screen to Printer

```
32000 REM * GR.0 SCREEN TO PRINTER *
32001 REM
32002 REM "OPEN" CLEARS SCREEN
32003 REM DO THIS EARLY IN PROGRAM
32004 REM USE GOSUB 32010 FOR THIS
32005 REM
32010 POKE 82,0:POKE 83,39
32020 OPEN #5,4,0,"S:":OPEN #7,8,0,"P
      :"
32030 RETURN
32031 REM
32032 REM USE GOSUB 32040 TO LPRINT
32033 REM TEXT FROM SCREEN
32034 REM
32040 PRINT #7;CHR$(10)
32050 FOR Y=0 TO 23
32060 FOR X=0 TO 39
32070 POSITION X,Y
32080 GET #5,G
32090 PUT #7,G
32100 NEXT X
32105 PRINT #7
32110 NEXT Y
32120 POSITION 0,0
32130 RETURN
```

1

GTIA: An Illustrated Overview

Louis and Helen Markoya

Type in these graphics demonstrations and see the startling displays made possible with the GTIA chip.

Have you ever seen computer-generated graphic displays that seem truly 3-D? The ones of landscapes or the ones of molecular structures? Have you ever wished you could generate similar graphics on your own machine? If so, and you own a 400 or 800, with the GTIA chip you're halfway there.

Modes 9, 10, and 11 are far from ordinary. They are called from BASIC by typing GRAPHICS 9, 10, or 11. They all offer the same screen resolution, 80 horizontal x 192 vertical, but different color selection.

GRAPHICS 9 offers *16* shades of any of the 16 colors, thus raising the machine's color capabilities to *256*. GRAPHICS 10 offers the programmer a choice of 9 of any of the 128 colors normally offered by the Atari, and GRAPHICS 11 gives the ability to present 16 different colors in any of the 8 luminances (shades). For those who had the machine before this new addition, the authorized Atari service center nearest you should be stocking this part.

GTIA is Atari's television Interface Chip. It is completely compatible with the hardware and software previously available. The only problem arises when software relying on the GTIA modes is run on a computer without this chip. Something will go to the screen, but not the desired effect.

GTIA is controlled for the most part by ANTIC, a microprocessor dedicated to the screen display. The GTIA processes digital commands from ANTIC or the 6502 (in the case of an interrupt) into the signal that goes to the television. GTIA also handles the tasks of color, player/missile graphics, and collision detection.

GTIA adds powerful capabilities in graphics modes 9, 10, and 11. All modes are extensions of graphics mode 8 + 16, ANTIC Mode 15. The display list remains the same, and the new modes are selected by the Priority Register. This Operating System Shadow Register, called PRIOR, is located at decimal 623, hex 26F. Bits 6 and 7 control the GTIA modes. When both are zero, GTIA works exactly the same as CTIA. When only bit 6 is set, GRAPHICS 9 is called; when only bit 7 is set, GRAPHICS 10 is called; and when both bits 6 and 7 are set, GRAPHICS 11 is called.

GRAPHICS 9

Setting bit 6 of PRIOR produces GRAPHICS 9, giving 16 luminances of one color. ANTIC provides the pixel data, and the background register, 712, is used to select your color (POKE 712, Color * 16 or SETCOLOR 4, Color,0). Each screen byte is broken in half for screen formatting. A display block is four pixels across by one pixel down. Each four bits represents 16 color choices. The number you choose (0-15) in your COLOR statement equates the luminance value you wish to use. Here's a simple BASIC program used to demonstrate this:

```
10 GRAPHICS 9:REM GRAPHICS MODE 9 (1
   6 SHADES OF ONE COLOR)
20 SETCOLOR 4,6,0:REM SET BACKGROUND
    REGISTER TO COLOR DESIRED (PURPL
   E)
30 FOR I=0 TO 15:REM SET UP VARIABLE
    FOR BOTH COLOR (SHADE) AND POSIT
   ION
40 COLOR I:REM SHADE OF COLOR
50 PLOT I,0:REM PLOT FROM UPPER LEFT
    CORNER
60 DRAWTO I,191:REM DRAWTO LOWER LEF
   T CORNER
70 NEXT I:REM NEXT SHADE AND NEXT LI
   NE
80 GOTO 80:REM HOLD SCREEN
```

The wide choice of luminances or shades available here will be particularly useful for shading objects to give the impression of bas-relief or the third dimension. With some background in perspective and lighting, a person could create scenes with a great illusion of depth, realistic or contrived.

GRAPHICS 10

GRAPHICS 10 is called when bit 7 of PRIOR is set to one and bit 6 to zero. This mode uses all nine of the Atari's color registers found at decimal 704-712 (hex 2C0 through 2C8). Any 9 of the 128 colors normally available to your computer could be used in this mode by simply POKEing the desired color (remember, 16 * Color + luminance) into each register or POKEing the desired color into the player/missile registers (704 through 707), using SETCOLOR statements for the playfield and background registers.

COLOR 0 represents the background and is located at decimal register 704, colors (for COLOR statements) 1-8 follow in order from 705-712. The big advantage to mode 10 is that any of the colors you choose can be changed independently of the others. For example, once a scene is created, you could change the color of the sky from dark to light blue very easily (FOR I = 128 TO 144: POKE 704,I: NEXT I). This will rotate the background color smoothly through its eight shades. You may wish to add a loop to delay the color change. Playfield or player/missile colors can be changed at any time. Also, special effects and animation could be achieved by rotating the values in all these registers.

Program 1 draws a border around the screen in eight colors (first register is used for background) and then rotates the colors to give a theater marquee effect. To display even more of this mode's capabilities, add the following lines:

```
185 A=96:REM SETS A VARIABLE FOR THE
    BACKGROUD
272 A=A+1:POKE 704,A:REM CHANGES BAC
    KGROUND COLOR
275 IF A=255 THEN A=1:REM ALLOWS ONL
    Y GOOD COLOR VALUES
```

These additional lines will rotate the background color through all its possibilities while the border is rotating.

GRAPHICS 11

Mode 11 operates similarly to mode 9. The difference is that only one luminance or shade is used, and a choice of all 16 colors is given. Bits 6 and 7 are set to one for this mode. Again, the background register is used for the colors, with ANTIC supplying the data. COLOR 0-15 relates exactly to the COLOR segment in the SETCOLOR command. To initiate this mode, you must choose the luminance or shade you want. The color would be set by your COLOR statement (SE. 4,0,0-15 Lum choice). The background is always COLOR 0 (black). This mode allows fine color blending to produce rainbow effects and therefore a wider color choice for picture making.

Program 2 draws a cross in 16 colors, again using a 1 x 1 display block, and then draws an ellipse in 16 colors around the center of the cross. This program shows the versatility of color use in mode 11. No longer are we restricted to horizontal screen architecture for extra color with Display List Interrupts.

GRAPHICS 9

Program 3 draws a landscape and a simple molecular structure floating high above it. This display truly gives the impression of depth and shows what can be done using light and shadow in Graphics mode 9.

These demos are only an introductory hint of the spectacular effects possible via GTIA. You could add more color to modes 9 and 11 by using players and missiles or create dramatic effects by switching between these modes (or with GRAPHICS 8) by POKEing PRIOR with the desired value.

Program 1. GRAPHICS 10 Demonstration

```
10 GRAPHICS 10
20 POKE 704,96:REM SETS BACKGROUND (
   COLOR 0, COLPMO) TO DARK PURPLE
30 POKE 705,22:REM SETS COLOR 1, COL
   PM1 TO YELLOW
40 POKE 706,38:REM SETS COLOR 2, COL
   PM2 TO YELLOW ORANGE
50 POKE 707,54:REM SETS COLOR 3,COLP
   M3 TO ORANGE
60 POKE 708,70:REM SETS COLOR 4, COL
   PF0 TO RED
70 POKE 709,86:REM SETS COLOR 5, COL
   PF1 TO PURPLE
80 POKE 710,104:REM SETS COLOR 6, CO
   LPF2 TO BLUE
90 POKE 711,120:REM SETS COLOR 7, CO
   LPF3 TO BLUE GREEN
100 POKE 712,180:REM SETS COLOR 8, C
    OLPF4 TO GREEN
110 FOR I=1 TO 64:REM SETS UP VARIAB
    LE FOR COLOR AND POSITION
```

```
120 C=C*(C<8)+1:COLOR C:REM CHANGES
    COLOR VALUE
130 PLOT I,I:REM START AT LEFT HAND
    CORNER
140 DRAWTO I,191-I:REM DRAWTO BOTTOM
     LEFT CORNER
150 DRAWTO 79-I,191-I:REM DR. BOTTOM
     RIGHT CORNER
160 DRAWTO 79-I,I:REM DR. TOP RIGHT
    CORNER
170 DRAWTO I,I:REM DR. TOP LEFT TO C
    OMPLETE BORDER
180 NEXT I
190 Z=PEEK(712):REM SETS Z EQUAL TO
    THE VALUE IN THE LAST REGISTER
200 POKE 712,PEEK(711):REM ROTATES V
    ALUES FROM 711 TO 712
210 POKE 711,PEEK(710):REM ROTATES V
    ALUES FROM 710 TO 711
220 POKE 711,PEEK(709):REM ROTATES V
    ALUES FROM 709 TO 710
230 POKE 709,PEEK(708):REM ROTATES V
    ALUES FROM 708 TO 709
240 POKE 708,PEEK(707):REM ROTATES V
    ALUES FROM 707 TO 708
250 POKE 707,PEEK(706):REM ROTATES V
    ALUES FROM 706 TO 707
260 POKE 706,PEEK(705):REM ROTATES V
    ALUES FROM 705 TO 706
270 POKE 705,Z:REM ROTATES VALUES FR
    OM 712 TO 705
280 FOR I=0 TO 15:NEXT I:REM SLOW DO
    WN ROTATION
290 GOTO 190:REM START AGAIN
```

Program 2. GRAPHICS 11 Demonstration

```
5 REM GRAPHICS 11 DEMONSTRATION PAGE
   5
10 A=1:R=26:REM SETS VARIABLES
20 DIM X(360),Y(360):REM ALLOW STORA
   GE SPACE FOR X AND Y COORDINATES
30 GRAPHICS 11:SETCOLOR 4,0,12:DEG :
   REM SETS GRAPHICS MODE, LUM OF CO
   LORS AND DEGREE MODE FOR ELIPSE
40 FOR I=0 TO 15:REM COLOR AND POSIT
   ION VARIABLE
50 COLOR I
60 PLOT 31+I,0:DRAWTO 31+I,191
```

```
70 PLOT Ø,86+I:DRAWTO 79,86+I:REM DR
   AWS CROSS
8Ø NEXT I
9Ø FOR I=Ø TO 36Ø STEP 2
1ØØ X(I)=R*COS(I)+34
11Ø Y(I)=R*SIN(I)+95
12Ø NEXT I:REM SETS X AND Y VALUES F
    OR PLOTTING ELIPSE
13Ø FOR I=Ø TO 36Ø STEP 2:REM CALLS
    ABOVE VALUES
14Ø COLOR A
15Ø PLOT X(I)+A,Y(I)+A:REM PLOT EACH
     COLORS' ELIPSE
16Ø NEXT I
17Ø A=A+1:REM NEXT COLOR AND NEXT EL
    IPSE POSITION
18Ø IF A=16 THEN 2ØØ:REM END IF ALL
    COLORS ARE USED
19Ø GOTO 13Ø:REM DRAW NEXT ELIPSE
2ØØ GOTO 2ØØ
```

Program 3. GRAPHICS 9 Demonstration

```
1Ø R=16:X=Ø:C=15
2Ø GRAPHICS 9:SETCOLOR 4,13,Ø
3Ø FOR I=13Ø TO 191
4Ø COLOR C
5Ø PLOT Ø,I:DRAWTO 79,I
6Ø X=X+1:IF X=4 THEN X=Ø:C=C-1
7Ø NEXT I
8Ø FOR I=Ø TO 79 STEP 8
9Ø COLOR 3:PLOT 59,13Ø:DRAWTO I,191
1ØØ NEXT I
11Ø COLOR 1:FOR I=Ø TO 7:PLOT 2,164:
    DRAWTO 21,158+I:NEXT I
12Ø COLOR 15:FOR I=Ø TO 3:PLOT 21,14
    Ø:DRAWTO 21+I,164-I*2:NEXT I
13Ø COLOR 4:FOR I=Ø TO 4:PLOT 2Ø,14Ø
    :DRAWTO 17+I,16Ø+I:NEXT I
14Ø FOR Z=1 TO 15
15Ø FOR I=Ø TO 36Ø STEP 6
16Ø X=Ø.25*R*COS(I)+35
17Ø Y=R*SIN(I)+5Ø
18Ø COLOR Z
19Ø PLOT X,Y
2ØØ PLOT X+1Ø,Y+17
21Ø PLOT X+3Ø,Y-2Ø
22Ø PLOT X-2,Y+12
23Ø PLOT X+21,Y+7Ø
```

```
240 NEXT I
250 R=R-1
260 NEXT Z
270 FOR I=2 TO 4:COLOR I:PLOT 46,72:
    DRAWTO 51+I,106
280 PLOT 43,62:DRAWTO 39,50+I
290 PLOT 47,62:DRAWTO 60+I,35
300 NEXT I
310 GOTO 310
```

2
Colors

Using SETCOLOR, COLOR, and POKE to Color Your Screen

Stephen Levy

The SETCOLOR and COLOR statements are important to Atari graphics, but are often misunderstood by beginners. Here are simple examples to help newcomers get the most from their Ataris.

One confusing aspect of Atari graphics for the beginner is the relationship between the COLOR and SETCOLOR statements. Once the relationship between these statements is understood, the beginner can control the color patterns he or she wishes to create.

SETCOLOR in Mode 0

The SETCOLOR statement is used to define the colors which will appear on the screen. In mode 0 the statement will function alone. Try entering the following without a line number:

```
SETCOLOR 2,0,14 〈RETURN〉
```

The screen should have turned white. But characters on a white screen are difficult to read. To remedy this situation, enter the following line:

```
SETCOLOR 1,0,4 〈RETURN〉
```

The print on the screen is now dark enough to read easily. To try one more color change, enter this line exactly as it's printed:

```
SE. 4,3,6 〈RETURN〉
```

SE. is the abbreviation for SETCOLOR and may be used to enter the statement, even within a program line. As an example, try entering this line using the SETCOLOR and GRAPHICS abbreviations as follows:

```
100 GR. 0:SE.2,8,6
```

Now list the line by typing L. ⟨RETURN⟩ (L. is the same as LIST). Notice that the computer has printed the entire word for SETCOLOR and GRAPHICS.

Take a closer look at the SETCOLOR statement. The first number is the color register, the second number is the hue, and the last number is the luminance. There are 16 hues to choose from and 8 luminances for each hue. Table 1 summarizes the hues.

Table 1. Color Numbers

Number	Color*	Number	Color
0	Gray	8	Light Blue
1	Gold	9	Blue-Green
2	Orange	10	Turquoise
3	Red-Orange	11	Green-Blue
4	Pink	12	Green
5	Violet	13	Yellow-Green
6	Purple-Blue	14	Orange-Green
7	Blue	15	Light Orange

*Colors may vary on different color TVs.

The simple examples of the SETCOLOR statement given previously used registers 1, 2, and 4. The use of these registers varies for the different modes. In mode 0, SETCOLOR 1 controls the print color and should have the same hue as register 2, which controls the screen color. Register 4 controls the border color for mode 0. Table 2 summarizes which color registers are used in which modes. It is important to remember that the term *color register* refers to the first number in the SETCOLOR statement, not to the first number in the COLOR statement, which is discussed below.

SETCOLOR and COLOR in Modes 3, 5, and 7

Look at Program 1. Lines 20, 30, and 40 set the color registers 0, 1, and 2. But just setting the registers does nothing to put color on the screen. A COLOR statement is needed to tell the computer which of the colors you wish to use, and something must be PLOTted or drawn.

The numbering can be a bit confusing. Refer to the first two columns of Table 2. Notice that to get the color established by the SETCOLOR 0,4,6 statement, you must issue the command COLOR 1. The COLOR statement number is always one more than the first number in the SETCOLOR statement, except for SETCOLOR 4, when COLOR 0 would be used.

If you haven't already typed in Program 1, do so now. Line 50 is the COLOR statement that decides which of the three SETCOLOR statements will be used when the box is drawn. This program will continue to rotate through the three colors endlessly until the BREAK key is pressed. Experiment with the program, change the colors, and see what happens.

Changing the Background Color

Program 2 illustrates how to change the background color in modes 3, 5, and 7. SETCOLOR 4 has just been hinted at so far. It is the color register that controls the background color in each of the four-color modes (3, 5, and 7).

Line 50 sets the background color. Lines 20, 30, and 40 set the colors for the rectangles. The text window keeps you informed about the colors. Again, the best way to learn about these colors is to experiment.

POKE Instead of SETCOLOR

Replace lines 20, 30, 40, and 50 of Program 2 with those lines from Program 3. Once this is done, run the program. All that was done was to replace each of the SETCOLOR statements with a POKE statement. The new lines act exactly the same as the originals.

Here's how to figure the second number in the POKE statement: multiply the hue by 16 and add the luminance. For example, SETCOLOR 0, 3, 10 would become 3 times 16 (48) plus 10, for a total of 58 — POKE 708, 58. Table 2 shows which POKE location to use for each color register.

Table 2. SETCOLOR, COLOR, and POKE

SETCOLOR*	COLOR	POKE†	MODES
0,*a*,*b*	1	708,*c*	1-7
1,*a*,*b*	2	709,*c*	0,1,2,3,5,7
2,*a*,*b*	3	710,*c*	0-8
3,*a*,*b*	-	711,*c*	1 and 2
4,*a*,*b*	0	712,*c*	0-8

**a* = any number from 0 to 15.
b = zero or any even number from 2 to 14.
†*c* = *a* x 16 + *b* (*a* times 16 plus *b*).

Selecting the Right Colors

One problem that many new programmers have is selecting the best combination of colors. Each time a color is added or changed on the screen, it affects the other colors on the screen. Program 4, a simple color editor, is intended to help with this problem. It will tell you the number to POKE to get the color you have selected. It will also tell you which color number to use in the COLOR statement.

Once the program is typed in and run, you will see one number flashing. That is the current register that will be affected by the joystick plugged into port one. Moving the joystick forward advances the numbers, and pulling back makes the numbers decrease. Pressing the fire button will change the register that is affected by the joystick. Trial and error will be the best teacher here. When you discover a combination of colors you believe would fit your application, make a note of it so you can use that combination in your own program.

Only the Beginning

This article is just an introduction to the world of Atari graphics. There is much more to be learned, but with the simple techniques shown here, you should be able to make some very impressive displays.

Program 1. SETCOLOR Example

```
10 GRAPHICS 7:COLR=1:POKE 752,1
20 SETCOLOR 0,4,6
30 SETCOLOR 1,13,10
40 SETCOLOR 2,8,4
50 COLOR COLR
60 PRINT "{CLEAR}{DOWN}{8 SPACES}COL
   OR ";COLR;"{4 SPACES}SETCOLOR ";C
   OLR-1
80 FOR R=10 TO 60
90 PLOT 50,R:DRAWTO 110,R
100 NEXT R
110 COLR=COLR+1:IF COLR=4 THEN COLR=
    1
120 FOR DELAY=1 TO 250:NEXT DELAY
130 GOTO 50
```

Program 2. Background Colors

```
10 GRAPHICS 5:C=0
20 SETCOLOR 0,3,10:REM COLOR 1
30 SETCOLOR 1,12,14:REM COLOR 2
40 SETCOLOR 2,8,8:REM COLOR 3
50 B=C:SETCOLOR 4,B,6
60 FOR COLR=1 TO 3
```

```
70 READ R:COLOR COLR
80 FOR D=Ø TO 7
90 PLOT R+D,3:DRAWTO R+D,36
1ØØ NEXT D:NEXT COLR
11Ø PRINT "{CLEAR}SE.Ø,3,1Ø
    {5 SPACES}SE.1,12,14{4 SPACES}SE
    .2,8,8"
12Ø PRINT " COLOR 1{7 SPACES}COLOR 2
    {7 SPACES}COLOR 3"
13Ø TRAP 15Ø:PRINT "{DOWN}SE.4,";B;"
    ,6";"{3 SPACES}NEW BACKGROUND CO
    LOR";:INPUT C
14Ø C=INT(C):IF C<Ø OR C>15 THEN C=B
15Ø RESTORE :GOTO 5Ø
16Ø DATA 9,37,65
```

Program 3. POKE Instead of SETCOLOR

```
2Ø POKE 7Ø8,58:REM COLOR 1
3Ø POKE 7Ø9,2Ø6:REM COLOR 2
4Ø POKE 71Ø,136:COLOR 3
5Ø B=C:POKE 712,B*16+6
```

Program 4. Color Editor

```
1Ø GRAPHICS 3+16
2Ø START=PEEK(56Ø)+PEEK(561)*256+4
3Ø POKE START-1,71
4Ø FOR R=2 TO 4:POKE START+R,7:NEXT
   R
5Ø POKE START+21,65:POKE START+22,PE
   EK(56Ø):POKE START+23,PEEK(561)
6Ø POKE 87,3:POKE 712,1Ø:B=6
7Ø FOR A=1 TO 3
8Ø COLOR A
9Ø FOR R=9 TO 18
1ØØ PLOT A+B,R:DRAWTO A+1Ø+B,R
11Ø NEXT R
12Ø B=B+1Ø
13Ø NEXT A
14Ø POKE 87,2:DIM BL$(3):BL$="
    {3 SPACES}"
15Ø POSITION Ø,Ø:PRINT #6;"712, 7Ø8,
     7Ø9, 71Ø,"
16Ø POSITION Ø,3:PRINT #6;"C.#"
17Ø POSITION 5,3:PRINT #6;"ONE  two
     THREE"
18Ø CL=3:SØ=7Ø:S1=15Ø:S2=2Ø6:S4=1Ø
19Ø GOSUB 27Ø:GOSUB 3ØØ:GOSUB 33Ø:GO
    SUB 34Ø
```

```
200 IF STRIG(Ø)=Ø THEN CL=CL+1
210 FOR W=1 TO 1ØØ:NEXT W
220 IF CL=4 THEN CL=Ø
230 ST=STICK(Ø):IF ST<>15 THEN POKE
    77,Ø
240 ON CL+1 GOSUB 25Ø,28Ø,31Ø,34Ø:GO
    TO 2ØØ
250 IF ST=14 THEN SØ=SØ+2:IF SØ>255
    THEN SØ=Ø
260 IF ST=13 THEN SØ=SØ-2:IF SØ<Ø TH
    EN SØ=254
270 POSITION 5,1:GOSUB 37Ø:POSITION
    5,1:PRINT #6;SØ:POKE 7Ø8,SØ:RETU
    RN
280 IF ST=14 THEN S1=S1+2:IF S1>255
    THEN S1=Ø
290 IF ST=13 THEN S1=S1-2:IF S1<Ø TH
    EN S1=254
300 POSITION 1Ø,1:GOSUB 37Ø:POSITION
     1Ø,1:PRINT #6;S1:POKE 7Ø9,S1:RE
    TURN
310 IF ST=14 THEN S2=S2+2:IF S2>255
    THEN S2=Ø
320 IF ST=13 THEN S2=S2-2:IF S2<Ø TH
    EN S2=254
330 POSITION 16,1:GOSUB 37Ø:POSITION
     16,1:PRINT #6;S2:POKE 71Ø,S2:RE
    TURN
340 IF ST=14 THEN S4=S4+2:IF S4>255
    THEN S4=Ø
350 IF ST=13 THEN S4=S4-2:IF S4<Ø TH
    EN S4=254
360 POSITION Ø,1:GOSUB 37Ø:POSITION
    Ø,1:PRINT #6;S4:POKE 712,S4:RETU
    RN
370 PRINT #6;BL$:FOR W=1 TO 15:NEXT
    W:RETURN
```

Rainbow Graphics

John R. Slaby

Try these variations and see if you can't make use of this new technique in games and graphics.

Here's how to print out any message in either GRAPHICS 1 or 2 and get up to seven colors for each character at the same time. Each color is limited to one scan line. After typing in the program and running it, you will be prompted to choose a graphics mode (1 or 2). You will then be asked for the message you want to display.

Capital and lowercase letters will be displayed in their normal solid default colors of orange and light green. Inverse capital and lowercase letters will be displayed in multicolor and will rotate in a curtain effect in opposite directions. The maximum length of the message was set at 120 characters/spaces, but this can be altered by changing the MSS$ dimension number. The input of the message has margins set to stimulate the GRAPHICS 1 or 2 display so that you don't end up with unintentionally hyphenated words.

The REMs in the program describe what is done where, but I believe three sections deserve some amplification. First, look at the 400 line numbers. The two FOR/NEXT loops store the various color values into the page six table locations for use by the DLI (the Display List Interrupt) and the vertical blank. Note that the color and luminance values increase for each location, thus giving a color and brightness change for each scan line.

If you want, you can easily change these values to give different effects. Choosing several colors and alternating them will give a barber pole effect. Keeping the same luminance value (make the I*2 just I) gives a sharper contrast between the colors. And to get a wider range of colors, change the color number by more than one for each scan line (for example, I*16*1.2). There are a great number of possibilities you can play with to get the effect you want; and since there are two tables, the two effects can be drastically different.

The vertical blank routine is contained in the DATA statements of the 300 lines. Its function is to rotate the two color tables in different directions to give the rolling curtain effect. If you want a static color display, you can eliminate the vertical blank by removing line 520 and adding POKE 54286, 192 to line 510.

Another easy variation is to change the rotation rate. The number 4 after the number 201 in line 320 controls the rate of change. Decrease this number to increase the rotation rate; increase it to slow it down. Please note that the maximum number is 255. If you exceed this number, you will register an error, which will be caught by the TRAP (line 50); this process will start the program over again and again if you don't correct the bad POKE value.

Multicolors with DLIs

The DLI is the key to the multicolors. In *Space Invaders* the DLIs are used to change the color of the invaders for each mode line. For GRAPHICS 0 and 1, the mode line consists of 8 scan lines, and for GRAPHICS 2, 16 scan lines. Normally, if you want to change colors for a mode line, you load a color value into the A, X, or Y registers, wait for horizontal sync (WSYNC), and then load the registers into the desired color registers during the horizontal blank. The problem is that the DLI is for the entire 8 or 16 scan lines, not each line.

Therefore, changing the color register after the horizontal blank results in your not knowing when the color change will take effect (for example, halfway across the scan line). One method to get around this would be to accurately count the 6502's cycles. When I first looked at the problem, this seemed the only way out; but not feeling that ambitious, I put this project on hold until I came up with an easier way.

The key is the WSYNC. For a meaningful display to exist on the TV screen, the WSYNC must occur every scan line, not every mode line. Thus, once you get control via the DLI, you keep it for eight WSYNCs and change the color registers during each scan line's horizontal blank. In theory, you can get eight colors for each character; but since the first line and the last lines are usually blank, six colors is what you actually get. If you want to modify the DLI, you could get 12 colors for GRAPHICS 2.

The one drawback of this method: if it is used for every mode line, it could tie up the 6502 processor for a good part of the available time. The majority of the remaining time would be during vertical blank. This would thus restrict the number of additional calculations, etc., you might want to do. This doesn't mean the technique is useless; it can be used for an eye-catching title page or used sparingly for a graphics display. It can also be used in games.

I believe this method is used in *Demon Attack* by Imagic for the Atari VCS. This game appears to change even the color of the player. You can also do this with my program by loading the player color register instead of the playfield. Overall, greater study of this method should allow programmers more latitude in creating striking visual effects.

The DLI and the vertical blank are not directly relocatable, but if you have any machine language capabilities, you should be able to modify them with little effort.

Rainbow Graphics

```
30 REM CHOOSE GRAPHICS MODE
40 DIM MSS$(120)
50 TRAP 1000
60 ? "WHAT GRAPHICS MODE? 1 OR 2";:
   INPUT A:IF A<>1 AND A<>2 THEN 60
70 GOSUB 100:GRAPHICS A+16
80 DL=PEEK(560)+256*PEEK(561)+6
90 FOR I=0 TO 6:POKE DL+I,133+A:NEX
   T I:GOTO 200
100 REM MODIFY MARGINS TO MAKE MESS
    AGE{8 SPACES}READABLE
110 POKE 82,10:POKE 83,29:?
120 REM INPUT MESSAGE
130 ? :? "WHAT IS MESSAGE?":? "USE
    ATARI INVERSE{3 SPACES}FOR COLO
    R EFFECT.{3 SPACES}CAPITAL OR L
    OWER."
140 INPUT MSS$:RETURN
200 REM LOCATE DLI
210 FOR I=0 TO 42:READ B:POKE 1553+
    I,B:NEXT I
220 DATA 72,138,72,152,72,162,0,141
    ,10,212
230 DATA 189,1,6,188,9,6,141,24,208
    ,140,25,208
240 DATA 232,173,0,6,201,1,144,3,14
    1,10,212
```

```
250 DATA 224,6,144,226,104,168,104,
    170,104,64
300 REM LOCATE DEFERED VERTICLE BLA
    NK
310 FOR I=0 TO 73:READ C:POKE 1599+
    I,C:NEXT I:POKE 1596,0
320 DATA 72,138,72,173,60,6,201,4,1
    44,55,162,7
330 DATA 173,8,6,141,61,6,173,9,6,1
    41,62,6
340 DATA 202,189,1,6,157,2,6
350 DATA 224,0,208,245,173,61,6
360 DATA 141,1,6,162,0,189,10,6,157
    ,9,6,232,224,7,208,245,173,62,6
    ,141,16,6,169,0
370 DATA 141,60,6,238,60,6,104,170,
    104,76,98,228
400 REM COLOR DATA & # LINES FOR MO
    DE
410 FOR I=0 TO 7:POKE 1537+I,(8+I)*
    16+I*2:NEXT I
420 FOR I=0 TO 7:POKE 1545+I,(1+I)*
    16+I*2:NEXT I
430 POKE 1536,A
490 POSITION 0,0:? #6;" "
500 REM ENABLE DLI AND DVVBLK
510 POKE 512,17:POKE 513,6
520 POKE 548,63:POKE 549,6:POKE 542
    86,192
600 POSITION 1,2:? #6;MSS$
610 GOTO 610
1000 TRAP 40000:GOTO 50
```

Colors by Page Flipping

Robert W. Myers

"Colors by Page Flipping" allows you to mix colors to produce new ones.

Have you ever wanted more colors than are provided on your Atari? This program uses four colors in Graphics mode 2, and mixes them two at a time to produce a total of ten different colors.

Blending Colors

All this color, like almost everything on the TV screen, is really an illusion. The blending of colors takes place because the displays are changed back and forth so fast that our eyes cannot keep up with the changes. Therefore, we see only one color, which is a mixture of the colors in all the different displays. You can mix more than two colors at a time, but as the number of displays increases, the amount of flicker on the screen increases too. The practical limit is four displays mixing at once.

This mixing is done by using multiple screen RAM areas and changing the Load Memory Scan (LMS) bytes in the display list during the Vertical Blank Interrupt (VBI). I realize that this sounds like a very complicated thing to do, but it's not.

Understanding the Display List

The Display List is a program for the ANTIC chip, the microprocessor that controls the TV screen so that the 6502 is free to spend more of its time doing computational chores. The Display List is in RAM, and the first byte of the Display List can be found at PEEK (560) + 256*PEEK (561).

Usually you will find that the first three bytes are the code that causes the black area at the top of the screen (to insure that nothing is lost due to overscan of the TV). The next byte is the LMS byte which sets the D6 bit (64 decimal). Added to this 64 is the ANTIC graphics mode number, which is given in the table.

The LMS is a three-byte instruction. The 64 + mode # is the first byte; the second and third bytes are the address of the beginning of screen RAM.

This address is what we are interested in here. Rapidly changing it allows us to switch from one picture to another and back. We cannot do this address swapping from BASIC; it is far too slow. The LMS bytes are changed by a short machine language routine that is run 60 times a second while the picture is blanked out as it returns to the top of the screen to begin the next frame. This is *Vertical Blank Interrupt.*

The routine loads the LMS bytes with the address of the first (normal) screen RAM, then it does an exclusive-OR with one of the memory locations. This causes the memory location to toggle between 0 and 1. This 0 or 1 is used to determine whether a branch will be taken or not. If the branch is taken, the next instruction is JMP $E462, which puts the interrupt back in normal operation. If the branch is not taken, the LMS bytes are changed to the address of the other (alternate) screen RAM. Then comes the JMP $E462.

Using VBI

The VBI is amazingly easy to use. All you do is write your routine that is to run during the interrupt. Then you write a machine language program that puts the high byte of your routine's address into the X-register, the low byte into the Y-register, and the number 7 into the accumulator. Finally, you JSR $E45C. This second machine language program is at lines 160, 170, and 180 of my program.

After setting up your VBI to change the LMS, you print or plot and move one set of your screen RAM to the other (alternate) location that you have specified to the LMS. This technique should be usable with any multicolor display mode or any combination of display modes, not only to mix colors, but also to mix text and graphics, to display mixed resolutions, etc.

ANTIC Graphics Mode Numbers

BASIC mode	0	1	2	3	4	5	6	7	8
ANTIC mode	2	6	7	8	9	10	11	13	15

Mixing Colors

```
1 REM ****************************
2 REM *{25 SPACES}*
3 REM *  MIXING COLORS TO MAKE  *
4 REM *{4 SPACES}AN ATARI RAINBOW
  {5 SPACES}*
5 REM *{25 SPACES}*
6 REM ****************************
9 REM
10 PRINT CHR$(125):GRAPHICS 2+16:BRE
   AK=1000
15 REM MACHINE LANGUAGE TO BE RUN DU
   RING VERTICAL BLANK{9 SPACES}INTE
   RRUPT
20 FOR I=0 TO 36:READ A:POKE 1536+I,
   A:NEXT I
30 DATA 173,39,6,141,49,6,173,40,6,1
   41,50,6,173,51,6,73,1,141,51
40 DATA 6,240,12,173,41,6,141,49,6,1
   73,42,6,141,50,6,76,98,228
45 REM FIND DISPLAY LIST IN RAM
50 DLIST=PEEK(560)+256*PEEK(561)
55 REM MODIFY MACHINE LANGUAGE PROGR
   AM BY POKEING IN ADDRESSES  FROM
   DISPLAY LIST
60 BYTE=DLIST+4:GOSUB BREAK:REM LOAD
    MEMORY SCAN LOW BYTE
70 POKE 1540,LOW:POKE 1562,LOW
80 POKE 1541,HIGH:POKE 1563,HIGH
90 BYTE=DLIST+5:GOSUB BREAK:REM LOAD
    MEMORY SCAN HIGH BYTE
100 POKE 1546,LOW:POKE 1568,LOW
110 POKE 1547,HIGH:POKE 1569,HIGH
120 BYTE=DLIST+20:GOSUB BREAK:REM NO
    RMAL SCREEN RAM
130 POKE 1576,HIGH:POKE 1575,LOW
140 BYTE=DLIST-250:GOSUB BREAK:REM A
    LTERNATE SCREEN RAM
150 POKE 1578,HIGH:POKE 1577,LOW
155 REM MACHINE LANGUAGE PROGRAM TO
    INITIALIZE VERTICAL BANK
    {4 SPACES}INTERRUPT
160 FOR I=0 TO 10:READ A:POKE 1600+I
    ,A:NEXT I
170 DATA 104,162,6,160,0,169,7,32,92
    ,228,96
180 X=USR(1600)
220 REM DRAW FIRST SCREEN
```

```
230 BYTE=DLIST-250:GOSUB BREAK
235 POKE 88,LOW:POKE 89,HIGH
240 POSITION Ø,4
250 PRINT #6;"ATARi coMPUTER Club"
260 PRINT #6
270 PRINT #6;"{4 SPACES}OF CHARLottE
    "
280 BYTE=DLIST+20:GOSUB BREAK
290 POKE 88,LOW:POKE 89,HIGH
305 REM SETCOLORS AND DRAW SECOND SC
    REEN
312 SETCOLOR Ø,12,6
313 SETCOLOR 1,4,6
314 SETCOLOR 2,15,8
315 SETCOLOR 3,8,6
320 POSITION Ø,4
330 PRINT #6;"AtaRi COMPUTeR ClUB"
340 PRINT #6
350 PRINT #6;"{4 SPACES}OF CHArlOttE
    "
359 REM HOLD IMAGE ON SCREEN
360 GOTO 360
999 REM SUBROUTINE TO BREAK DOWN NUM
    BER INTO HIGH AND LOW BYTES
1000 HIGH=INT(BYTE/256)
1010 LOW=BYTE-HIGH*256
1020 RETURN
```

2

A Fill-In on XIO(FILL)

Gretchen Schabtach

This tutorial article presents several interesting extensions for the XIO(FILL) program on page 54 of the Atari BASIC Reference Manual.

Your Atari readily fills line drawings of figures with color using a special application of the XIO statement. However, the example in the *Atari BASIC Reference Manual* (page 54) can be expanded to demonstrate the strengths and limitations of this application. A critical point about XIO filling is that filling stops when a pixel previously filled with color is encountered. Thus, interesting effects occur when the Atari is commanded to fill overlapping figures with color.

To get the most from the following short programs, begin by running the demonstration program on page 54 of the *BASIC Reference Manual.* Then run Program 1. Program 1 generates three rectangles, randomly positioned, with random proportions, and fills them from top to bottom and left to right with three different colors. Observe what happens when the figures overlap.

Moving line by line, from top to bottom and left to right, the fill stops when a colored pixel is encountered. Thus, when the program generates two overlapping rectangles, filling of the second rectangle stops whenever the first filled rectangle is encountered—and does not resume even if the second rectangle extends to the right beyond the first rectangle.

With a few modifications, Program 1 is not only illuminating with regard to the XIO(FILL) function, but also much more interesting. First, randomly change the colors to be used in filling. Second, generate rectangles continuously. To do this, make the following changes:

```
 20 N = INT(RND(0)*3+1)
160   (i.e., delete)
170 GOTO 20
```

After you've modified the program as specified above, run it and admire the changes. Now, add a little music to your life. This is easily accomplished by adding the following statements:

81 I = INT(RND(0)*256)
82 SOUND 1,I,10,4
84 SOUND 2,255-I,10,4

Try this, and then delete line 150. This will speed things up a little and make them even more interesting.

Finally, black backgrounds can become tiresome. To randomly change the background color, type in the following:

11 REM : CHANGE BACKGROUND
12 B = INT(RND(0)*16)
13 SETCOLOR 4,B,2
166 REM : CHANGE BACKGRD SO GOTO 12
170 GOTO 12

Your final program listing should look like that shown in Program 2.

There are further simple and interesting modifications. For example, vary the constants in statements which include random number generators—those containing (RND(0)); or delete + 16 in line 10 and provide yourself with a text window in which you can write commentary on what the viewer sees; or change the characteristics of the shapes generated to be filled with color (lines 40 through 80). Your imagination will suggest other possibilities.

Program 1. XIO Example 1

```
1 REM : DEMO OF XIO FILLING
2 REM : SUPPRESS WINDOW IN GR.7
10 GRAPHICS 7+16
15 REM : ESTABLISH 3 FILL COLORS
20 FOR N=1 TO 3 STEP 1
30 COLOR N
35 REM : GENERATE FIGURE TO FILL
40 X1=INT(RND(0)*80)
50 Y1=INT(RND(0)*48)
60 X2=X1+INT(RND(0)*80)
70 Y2=Y1+INT(RND(0)*48)
80 IF X1=X2 OR Y1=Y2 THEN 40
90 PLOT X2,Y2
100 DRAWTO X2,Y1
110 DRAWTO X1,Y1
115 REM : FILL FIGURE
120 POSITION X1,Y2
```

```
130 POKE 765,N
140 XIO 18,#6,0,0,"S:"
150 FOR W=1 TO 400:NEXT W
155 REM : CHANGE COLOR FOR NEXT FIGU
    RE
160 NEXT N
165 REM :GENERATE NEW FIGURE
170 GOTO 10
```

Program 2. XIO Example 2

```
1 REM : DEMO OF XIO FILLING #2
5 REM : SUPPRESS WINDOW IN GR.7
10 GRAPHICS 7+16
11 REM :CHANGE BACKGROUND
12 B=INT(RND(0)*16)
13 SETCOLOR 4,B,2
15 REM : ESTABLISH 3 FILL COLORS
20 N=INT(RND(0)*3+1)
30 COLOR N
35 REM : GENERATE FIGURE TO FILL
40 X1=INT(RND(0)*80)
50 Y1=INT(RND(0)*48)
60 X2=X1+INT(RND(0)*80)
70 Y2=Y1+INT(RND(0)*48)
80 IF X1=X2 OR Y1=Y2 THEN 40
81 I=INT(RND(0)*256)
82 SOUND 1,I,10,4
84 SOUND 2,255-I,10,4
90 PLOT X2,Y2
100 DRAWTO X2,Y1
110 DRAWTO X1,Y1
115 REM : FILL FIGURE
120 POSITION X1,Y2
130 POKE 765,N
140 XIO 18,#6,0,0,"S:"
155 REM : CHANGE COLOR FOR NEXT FIGU
    RE
165 REM :GENERATE NEW FIGURE
166 REM :CHANGE BACKGRD SO GOTO 12
170 GOTO 12
```

3

Redefining Character Sets

3

Character Generation

Charles Brannon

The ability to redefine characters is a very powerful feature of the Atari. This discussion explains how it is done.

Atari computers are among the few that possess a very powerful feature—the ability to redefine the character set. The *character set* is the group of 255 alphanumeric characters that can appear on the screen. It comprises the upper- and lowercase alphabet, the numbers, special symbols, and punctuation. Also included in the Atari character set are 29 "control graphics" characters. When the CTRL key is held down and a letter of the alphabet is typed, the corresponding graphics symbol is displayed. These symbols are much like those found on the PET/CBM. Unlike the PET, however, the Atari can redefine any of these characters. This allows custom graphics, user-defined special symbols (like pi, theta, or foreign language alphabets), and logos.

There is no built-in command to perform the changes; it has to be done the hard way with PEEK and POKE. These are commands to look at and modify memory, respectively. First of all, you must understand how the Atari stores and displays these characters. It is beneficial if you know how to work with binary numbers, but it is not a prerequisite.

Start out by designing your characters. Fill in the blocks on an 8x8 grid; each block will represent a pixel (picture element, or screen dot). Observe the A in Figure 1. Notice the heavy vertical lines. A television screen will display horizontal lines brighter than vertical lines, so it is necessary to have two vertical lines in order for it to be clearly visible (this also prevents "artifacting"). Therefore, the pi in Figure 2 may be hard to see unless enlarged in mode 1 or 2.

After you have designed your characters, you have to convert them to the numbers that a computer loves. Each row in your grid represents a binary *byte*. A filled-in block represents a 1 and a blank block means 0. Hence, the top row of the A is 00011000 or 24 decimal. Now write the bytes for each row. If you do not work with binary numbers, you can convert each line in the following manner:

1. Notice the numbers above each column. They are the powers of two.
2. If a block is filled in, take a number above it and add it to a "Sum." Sum up all the blocks in the row (for example, the fourth line of the pi would be 128 + 32 + 4 = 164).
3. Do this for all eight rows.

Figure 1. Character A

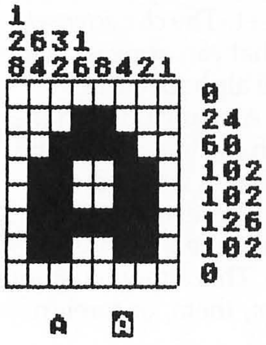
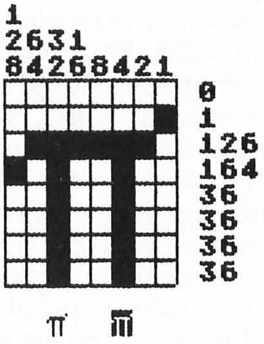


Figure 2. Character Pi



Next, assemble the numbers into DATA statements. The numbers for pi would then look like this:

```
1000 DATA 0,1,126,164,36,36,36,36
```

Finally, you have your numbers. Now all you have to do is replace the numbers of the character you want to redefine with your numbers. Unfortunately, this table is stored in Read Only Memory (ROM), so it cannot be altered. The solution is to copy this table into Random Access Memory (RAM), which can be changed, and then tell the computer where you have moved the characters.

First, we have to find a safe place in RAM to hold the character set table. One solution is to place the character set at the top of memory. We can use memory location 106, which holds the number of the topmost page (a "page" is 256 bytes) of memory. On a 32K machine, this is usually 128 (128*256=32768=32*1024).

We would need to store our character set four pages beneath this limit, since we need 1024 bytes, and 4*256 = 1024.

However, there is a complication. The operating system has also decided to use the top of memory to store information needed to display the video screen. All we need to do is store our character set a little further back in memory, behind the screen. Here's the tricky part. Exactly how much we need to "step back" depends on which graphics screen is used, since different modes use varying amounts of memory (from 272 bytes in GRAPHICS 3, to nearly 8,000 bytes in GRAPHICS 8). Just remember to step back far enough to get past the screen, and make the "step size" a multiple of four.

So to find the place to store a character set with graphics modes 0,1,2,3,4 and 5, step back four pages to get past the screen, then step back four more to hold your character set, for a total of eight. GRAPHICS 6 would require 12, GRAPHICS 7 would require 20, and you would need to step back a whopping 36 pages to fit your character set beneath a GRAPHICS 8 display.

A sample program to redefine the character set (assuming you've set up your DATA statements) might start out like this:

```
10 CHBAS=57344
20 CHSET=(PEEK(106)-8)*256
30 FOR I=0 TO 1023
40 POKE CHSET+I,PEEK(CHBAS+I)
50 NEXT I
```

These lines transfer the ROM-based character set to RAM, where they can be modified. The transfer (in BASIC) takes a long time to execute, about 15 seconds. You don't need to copy the character set if you want to redefine the entire character set, or don't need any characters from the default character set. And unless you go into a higher graphics mode that might wipe out the character set, you don't need to execute these lines more than once. The data will still be there, even when you RUN the program again.

The next line:

```
60 POKE 756,CHSET/256
```

tells ANTIC (the Atari's video microprocessor) where you have placed your character set. This location normally contains 224 (224*256 = 57344, which is the address in ROM of the default character set).

Now that the table is in RAM, we can find the place in it for the new numbers. Look up the character you want to replace

(Table 9.6–Internal Character Set, in the *Atari BASIC Reference Manual*). Note that this number is *not* the ATASCII value of the character. Include this number preceding your eight bytes in the DATA statements. For our pi:

```
1000 DATA 32,0,1,126,164,36,36,36,36
```

The 32 is the internal code for the @ symbol. Anytime you press the @ key, you may be surprised to see pi.

A few more lines, and the program is finished:

```
70 READ NCHR:REM NUMBER OF CHARACTER
   S TO BE CUSTOMIZED
80 FOR I=1 TO NCHAR
90 READ RPLC:REM INTERNAL VALUE OF C
   HARACTER TO BE REPLACED
100 FOR J=0 TO 7
110 READ A
120 POKE CHSET+8*RPLC+J,A
130 NEXT J
140 NEXT I
150 REM LINE 170 IS OPTIONAL
160 REM IT JUST DISPLAYS ALL THE CHA
    RACTERS
170 FOR I=0 TO 255:PRINT CHR$(27);CH
    R$(I);:NEXT I
180 END
998 REM NUMBER OF CHARACTERS
999 DATA 1
```

A few program notes:

1. You can use multiple statements per line; delete REMs if you like.
2. This program should be appropriately renumbered, and RETURN added if you want to use it as a subroutine.
3. This is not the only way to customize the character set. See the other articles in this book for more ideas.

The complete program (less DATA statements) and a utility program that lets you look at characters wrap up this article. Study them, puzzle them out, and have fun with Atari's custom characters!

Program 1. Redefining Characters

```
10 CHBAS=57344
20 CHSET=(PEEK(106)-8)*256
30 FOR I=0 TO 1023
```

```
40 POKE CHSET+I,PEEK(CHBAS+I)
50 NEXT I
60 POKE 756,CHSET/256
70 READ NCHR:REM NUMBER OF CHARACTER
   S TO BE CUSTOMIZED
80 FOR I=1 TO NCHAR
90 READ RPLC:REM INTERNAL VALUE OF C
   HARACTER TO BE REPLACED
100 FOR J=0 TO 7
110 READ A
120 POKE CHSET+8*RPLC+J,A
130 NEXT J
140 NEXT I
150 REM LINE 170 IS OPTIONAL
160 REM IT JUST DISPLAYS ALL THE CHA
    RACTERS
170 FOR I=0 TO 255:PRINT CHR$(27);CH
    R$(I);:NEXT I
180 END
998 REM NUMBER OF CHARACTERS
999 DATA 1
1000 DATA 32,0,1,126,164,36,36,36,36
```

Program 2. View Characters

```
10 GRAPHICS 4
20 SCR=PEEK(88)+256*PEEK(89)
30 PRINT "{CLEAR}CHARACTER #? (0-127
   )";
40 INPUT CHR
50 IF CHR<0 OR CHR>127 THEN 30
60 PRINT #6;CHR$(125)
70 FOR I=0 TO 7
80 POKE SCR+4+10*I,PEEK(57344+CHR*8+
   I)
90 NEXT I
100 GOTO 30
```

3

Custom Characters

Charles Delp

Custom character graphics is an easy way to program game animation, but sometimes it results in uneven motion. Smoother animation can be achieved by using custom characters to create the fixed playfield, and then using player/missile graphics to animate the players. The three programs here show you how.

One of the easiest ways to put colorful, high-resolution playfields or special symbols on the screen is with character graphics, employing custom characters.

One of the major drawbacks of using character graphics to animate a game is that players can move only in large, character-sized jumps. When smoother action is desired, a better solution is to draw the fixed playfields using custom characters and then animate the players using player/missile graphics.

The advantages of using character graphics rather than bitmapped graphics to draw fixed playfields are:

1. Much less memory is required to achieve the same resolution.
2. More colors are available.
3. Less time is required to draw to screen memory.
4. Color fill is faster and easier.

The major disadvantage of using character graphics to draw fixed playfields is that only two colors (character color and background color) are available within any one character. The table shows the resolution, memory requirements, and colors available for various Atari BASIC character and bitmapped graphics modes.

How Characters Are Defined

Atari characters are defined by 64 pixels arranged in eight columns by eight rows. From right to left, the values of the columns are 1, 2, 4, 8, 16, 32, 64, and 128. If a particular pixel is turned on, the value of that column is added to the row total; if the pixel is turned off, zero is added to the row total. The total value of all the "on" pixels in a row forms a byte of data which defines that row. Each of the eight rows is defined by a byte of data, for a total of eight bytes per character. (See Figure 1 for a specific example.) Note how the row bytes are arranged in memory from the character start address (CHADD).

Atari Display Mode Facts

Graphics Mode	Graphics Type	Resolution H x V	Colors Available (Including Background)	Bytes of Memory/ Screen
0	Character	320 x 192	2	960
1	Character	160 x 192	5	480
2	Character	160 x 96	5	240
5	Bitmapped	80 x 48	4	960
7	Bitmapped	160 x 96	4	3840
8	Bitmapped	320 x 192	2	7680

Character Color Information

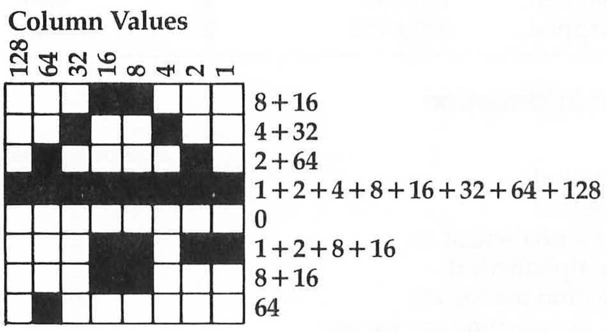
Character Type	Color Register
Uppercase alphabetical	0
Lowercase alphabetical	1
Inverse uppercase alphabetical	2
Inverse lowercase alphabetical	3
Numbers, punctuation marks, etc.	0
Inverse numbers, punctuation marks, etc.	2

Character Editor

Program 1 is a character editor utility which will be a help in developing the DATA statements required to define each character. Draw the character using the joystick. Erase errors by holding the trigger button while drawing over the error. Press C (Clear character) at any time to clear the screen for another character. Press D (Demonstrate character) to see the character in all three of the character graphics modes, as well as the DATA statement required to produce the character. Press P (Print data) to print a hard copy of the character DATA statement. Press E (Enter data) to enter the character data as a program line beginning at line 9000. When all characters have been entered, typing LIST "D:CHAR",9000,9999 will save the data to disk or LIST "C", 9000,9999, to cassette. The data may be merged into your graphics program using the ENTER command (see Chapter 5, *Atari BASIC Reference Manual).*

Figure 1. Typical Custom Character

Row Byte Memory Location	Rows
CHADD	Byte 1=24
CHADD +1	Byte 2=36
CHADD +2	Byte 3=66
CHADD +3	Byte 4=255
CHADD +4	Byte 5=0
CHADD +5	Byte 6=27
CHADD +6	Byte 7=24
CHADD +7	Byte 8=64



Locating the Custom Character Set in Memory

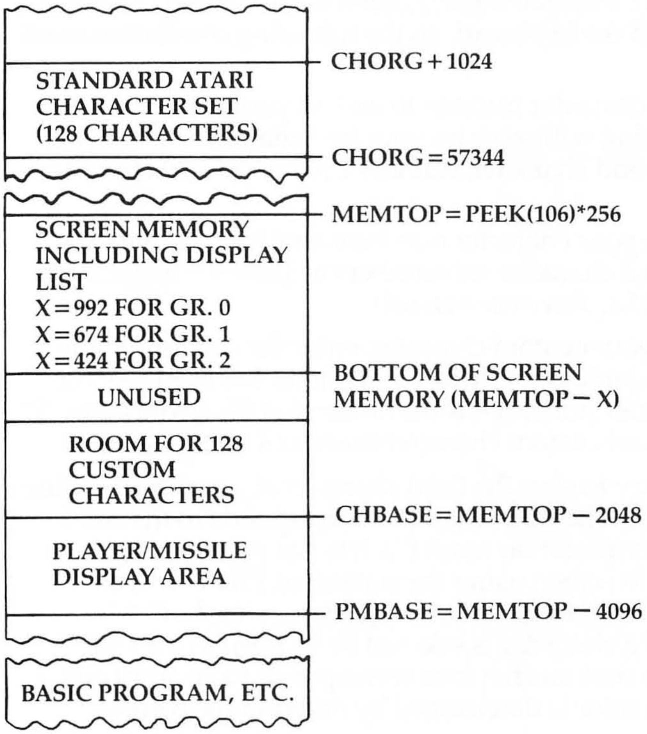
First, look at the memory map in Figure 2. The standard Atari character set is located in ROM beginning at address 57344 (CHORG). The location and size of screen memory including the display list will depend on how much RAM is installed in your computer and which graphics mode is called by your program. The new character set must be defined and stored in RAM in a location which does not interfere with screen memory, the display list, the player/missile display memory, or the BASIC program. The procedure described below and illustrated in Figure 2 will keep everything nicely separated.

1. Find MEMTOP on your computer by entering the following line: PRINT PEEK (106)*256.

2. Decide whether your program using the custom characters will be written in graphics mode 0, 1, or 2. For your information, the bottom of screen memory, including display list, will be located at MEMTOP – X
   where X = 992 for graphics mode 0
         X = 674 for graphics mode 1
         X = 424 for graphics mode 2

3. The starting address of the custom character set, CHBASE, must be located on a 1K memory boundary, so CHBASE should start 2K below MEMTOP (1K for screen memory, 1K for the character set); therefore, CHBASE = MEMTOP – 2048.
4. If player/missile graphics are to be used, PMBASE must be located on a 2K boundary (for single line resolution), so P/M Base should start 4K below MEMTOP (1K for screen memory, 1K for the character set, 2K for P/M Display Area); therefore, PM BASE = MEMTOP – 4096.

Figure 2. Memory Map


STANDARD ATARI CHARACTER SET (128 CHARACTERS)
CHORG + 1024
CHORG = 57344
SCREEN MEMORY INCLUDING DISPLAY LIST
X = 992 FOR GR. 0
X = 674 FOR GR. 1
X = 424 FOR GR. 2
MEMTOP = PEEK(106)*256
UNUSED
BOTTOM OF SCREEN MEMORY (MEMTOP – X)
ROOM FOR 128 CUSTOM CHARACTERS
CHBASE = MEMTOP – 2048
PLAYER/MISSILE DISPLAY AREA
PMBASE = MEMTOP – 4096
BASIC PROGRAM, ETC.

Developing a Custom Character Set

Normally a character set consists of 128 different characters in graphics mode 0, and 64 different characters in graphics modes 1 and 2. However, a character set need not be full and may contain only as many characters as needed to meet the requirements of your program. The first character in the set must always be a space (DATA statement filled with zeros).

Program 2 demonstrates how to set up and use a custom character set containing only custom characters. To keep things simple, the set contains only eight characters.

Lines 10-30 Initialize and find CHBASE
Lines 50-70 Clear space in memory for the custom character set
Lines 90-130 POKE the new characters into memory beginning at CHBASE
Lines 200-280 Contain the character data
Lines 300-360 Print the characters on the screen

The simplest way to print the custom characters to the screen is with the PRINT #6; statement; however, the custom characters are not shown on the keyboard, so the following correlation must be performed:

1. Assign a character number to each of your custom characters, beginning with zero for your first character, number 1 for your second character, number 2 for your third character, etc.

2. Correlate your character numbers, one for one, with the Atari internal character set numbers in Table 9.6 (page 55 of the *Atari BASIC Reference Manual).*

3. To print your custom character, enter the corresponding Atari character in your PRINT statement. For example, the Atari character number 4 is the dollar sign ($). PRINT #6; "$" will print your custom character number 4 on the screen.

It is necessary to skip the third character of your set. (See line 220 of Program 2.) The third character corresponds to the Atari internal character quotation mark ("). It is not possible to print a quotation mark to screen using the statement PRINT #6; and placing the quotation mark between quotation marks (" " ").

The color of a character is selected by its form in the PRINT statement. If the custom character corresponds to an Atari alphabetical letter, the color is determined by entering the corresponding Atari letter in the PRINT statement in upper- or lowercase, or inverse upper- or lowercase. Four colors are available for characters corresponding to Atari alphabetical letters.

If the custom character corresponds to an Atari number, punctuation mark, etc., the color is determined by entering the corresponding Atari number in the PRINT statement in standard or inverse video. The third and fourth colors for numbers and punctuation can be obtained by using control characters (see "Discovering 'Hidden' Graphics" in Chapter 1).

The PRINT #6; method of putting custom characters on the screen has some serious drawbacks. The method used in Program 3 may not be as easy to understand, but has fewer limitations, particularly for drawing entire playfields.

Mixing Standard and Custom Characters

In addition to colorful playfields, most games print numbers and specific letters on the screen to display such things as score, time, fuel, hits, etc. The standard Atari character set already contains these characters, so it would be pointless to develop custom characters for this purpose. The solution is to develop a custom character set containing all the necessary standard numbers and letters, but to replace all unneeded standard characters with custom characters.

The procedure for developing a mixed character set is described below:

Note: Refer to Table 9.6 in the *Atari BASIC Reference Manual,* page 55.

1. Determine which standard characters will be needed in your program.

2. Form a string variable which contains the unneeded standard characters. The string may include any unneeded characters with 0 and 127 for GRAPHICS 0, or between 0 and 63 for GRAPHICS 1 and 2. The only exception in either case is the quotation mark, for reasons explained before.

3. Copy the standard character set from CHORG (57344) to CHBASE by PEEK and POKE statements.

4. Modify the unneeded standard characters into custom characters by POKEing custom character data into the character address (CHADD) of each character in the string. (See Program 3 for specific details of the procedure.)

Printing Complete Playfields

Program 2 places the custom characters onscreen with PRINT #6; statements. A better, though more difficult, method is plotting the character on the screen using color data to designate which character is to be plotted and in what color the character will appear. The color data to define a character contains two elements: the character number (the Atari internal character set number from Table 9.6), and a plus or minus offset which determines the color of the character. The offsets may be obtained from Figure 9.7 on page 56 of the *Atari BASIC Reference Manual.* The easiest way to explain this concept is with examples.

Example 1: Suppose you want to display the standard character K in graphics mode 1 with color 0:

1. From Table 9.6, the internal character number for K is 43. Note that the K is from column 2.
2. From Table 9.7, the offset to produce a column 2 character in color 0 is + 32.
3. The color data to plot K in color 0 would be 43+32=75.
4. 210...
   220 Color 75
   230 Plot 5,7
   240...

The program lines above will print a K in color 0 at X=5, Y=7.

Example 2: Suppose you want to display your custom character number 19 in graphics mode 2 with color 3. Your character number 19 corresponds to the standard character ";":

1. From Table 9.6, the internal character number for the semicolon (;) is 27 from column 1.
2. From Table 9.7, the offset to produce a column 1 character in color 3 is + 128.
3. The color data to plot your custom character in color 3 would be 27+128=155.
4. 150...
   160 Color 155
   170 Plot 7,1
   180...

The program lines above will print your custom character in color 3 at X=7, Y=1.

A complete playfield may be drawn using the color/plot method by implementing a nested row, column loop which reads the color numbers from DATA statements and plots the characters to the screen (see lines 550 through 610 of Program 3 for a method).

Using the color data method has one other advantage: it allows easy printing of the quotation mark (") as well as all the numbers and punctuation in four colors.

Program 3 is a full screen, GRAPHICS 2, fixed playfield demonstration using 31 custom characters:

Lines 30-80	Initialize, define string, and find CHBASE.
Lines 110-130	Move standard character set down to CHBASE.
Lines 150-210	Modify the characters in the string into custom characters. Line 160 locates the correct addresses to modify. The −32 is an offset to change ATASCII to Atari internal character numbers.
Lines 301-331	Custom character data.
Line 420	Select split screen mode; kill cursor.
Lines 510-530	Change character set pointer; select colors.
Lines 550-610	Read color data and plot characters on screen.
Line 630	Print standard characters in text window.
Lines 650-680	Flicker engine exhaust.
Lines 700-709	Color data for 10 rows of 20 characters.

Program 1. Character Editor

```
5 CLR :? "{CLEAR}":OPEN #1,4,0,"K:":
  OPEN #6,4,0,"S:":SETCOLOR 2,9,2:SE
  TCOLOR 4,9,2:POKE 752,1
10 DIM C$(1),STORE(8):N=0
20 GOSUB 6000
40 ? :? :? "PLUG JOYSTICK INTO JACK
   1"
50 ? "DRAW CHARACTER WITH JOYSTICK"
60 ? "HOLD TRIGGER BUTTON TO ERASE":
   ? :?
70 ? :? :? "{8 SPACES}PLEASE WAIT .
   . ."
90 CHBASE=(PEEK(106)-8)*256:CHORG=57
   344
100 FOR I=0 TO 1023:POKE CHBASE+I,PE
    EK(CHORG+I):NEXT I
105 C$="&"
110 CHADD=CHBASE+(ASC(C$)-32)*8
120 POKE 756,CHBASE/256
200 ? "{CLEAR}":POKE 752,1:GOSUB 600
    0
205 FOR I=0 TO 7:STORE(I)=0:NEXT I
210 ?
```

```
220 ? "{5 SPACES}87654321"
230 ? "{4 SPACES}{Q}{8 R}{E}"
240 ? "{4 SPACES}|{8 SPACES}| 1"
250 ? "{4 SPACES}|{8 SPACES}| 2"
260 ? "{4 SPACES}|{8 SPACES}| 3
    {3 SPACES}C = CLEAR CHAR"
270 ? "{4 SPACES}|{8 SPACES}| 4
    {3 SPACES}D = DEMO CHAR"
280 ? "{4 SPACES}|{8 SPACES}| 5
    {3 SPACES}P = PRINT DATA"
290 ? "{4 SPACES}|{8 SPACES}| 6
    {3 SPACES}E = ENTER DATA"
300 ? "{4 SPACES}|{8 SPACES}| 7"
310 ? "{4 SPACES}|{8 SPACES}| 8"
320 ? "{4 SPACES}{Z}{8 R}{C}"
499 REM MAIN LOOP
500 X=7:Y=6
510 K=PEEK(764)
512 IF STRIG(0)=0 THEN 700
513 IF K=18 THEN 1000
514 IF K=58 THEN 2000
515 IF K=10 THEN 3000
516 IF K=42 THEN 5000
518 POSITION X,Y:? " "
520 FOR DELAY=1 TO 15:NEXT DELAY
530 POSITION X,Y:? "■"
540 FOR DELAY=1 TO 15:NEXT DELAY
550 ST=STICK(0)
560 IF ST=15 THEN 510
570 IF ST=6 OR ST=14 OR ST=10 THEN Y
    =Y-1
580 IF ST=5 OR ST=9 OR ST=13 THEN Y=
    Y+1
590 IF ST=5 OR ST=6 OR ST=7 THEN X=X
    +1
600 IF ST=9 OR ST=10 OR ST=11 THEN X
    =X-1
610 IF X>14 THEN X=14
620 IF X<7 THEN X=7
630 IF Y>13 THEN Y=13
640 IF Y<6 THEN Y=6
650 GOTO 510
700 POSITION X,Y:? "■"
710 FOR DELAY=1 TO 15:NEXT DELAY
720 POSITION X,Y:? " "
730 FOR DELAY=1 TO 15:NEXT DELAY
732 IF K=18 THEN 1000
734 IF K=58 THEN 2000
736 IF K=10 THEN 3000
```

```
738 IF K=42 THEN 5000
740 GOTO 550
999 REM CLEAR CHAR
1000 POKE 764,255
1010 GOTO 200
1999 REM DEMO CHAR
2000 BYTE=0:BIT=0
2005 GOSUB 4000
2008 REM DETERMINE DATA VALUES
2010 FOR Y=0 TO 7
2020 FOR X=7 TO 0 STEP -1
2030 LOCATE (X+7),(Y+6),PIX
2035 POSITION (X+7),(Y+6):PUT #6,PIX
2040 IF PIX=160 THEN PIX=1
2050 IF PIX=32 THEN PIX=0
2060 IF X=7 THEN BIT=PIX
2070 IF X=6 THEN BIT=PIX*2
2080 IF X=5 THEN BIT=PIX*4
2090 IF X=4 THEN BIT=PIX*8
2100 IF X=3 THEN BIT=PIX*16
2110 IF X=2 THEN BIT=PIX*32
2120 IF X=1 THEN BIT=PIX*64
2130 IF X=0 THEN BIT=PIX*128
2140 BYTE=BYTE+BIT
2150 NEXT X
2160 STORE(Y)=BYTE
2165 BYTE=0
2170 NEXT Y
2180 POSITION 2,16:? "DATA ";
2190 FOR Y=0 TO 6
2200 STORE=STORE(Y)
2210 ? STORE;",";
2220 NEXT Y
2230 STORE=STORE(7)
2240 ? STORE;
2242 FOR J=0 TO 7:STORE=STORE(J)
2244 POKE CHADD+J,STORE:NEXT J
2248 REM ALTER DISPLAY LIST
2250 A=PEEK(560)+PEEK(561)*256
2260 POKE A+25,6:POKE A+26,6:POKE A+
     27,7:POKE A+28,PEEK(A+29):POKE
     A+29,PEEK(A+30):POKE A+30,PEEK(
     A+31)
2265 REM PRINT CHAR TO SCREEN
2270 POSITION 2,18:? "{3 SPACES}GR 0
     :  & & & & & & & & & & & & &";
2280 POSITION 0,20:? #6;"GR 1: & & &
      & & & &";
2290 POSITION 0,21:? #6;"GR 2: & & &
      & & & &";
```

```
2345 POKE 764,255
2350 GOTO 500
2999 REM PRINT DATA TO PRINTER
3000 TRAP 3100
3005 POKE 559,0
3030 GOSUB 3200
3040 LPRINT "DATA ";S0;",";S1;",";S2
     ;",";S3;",";S4;",";S5;",";S6;",
     ";S7
3050 POKE 559,34
3060 POKE 764,255
3070 GOTO 200
3100 GOSUB 4000
3110 POKE 559,34
3120 POSITION 2,17
3130 ? "  PRINTER NOT CONNECTED"
3140 ? "{9 SPACES}- OR -"
3150 ? "{3 SPACES}PRINTER TURNED OFF
     "
3160 FOR DELAY=1 TO 400:NEXT DELAY
3165 GOSUB 4000
3170 POKE 764,255
3180 GOTO 200
3200 S0=STORE(0):S1=STORE(1):S2=STOR
     E(2):S3=STORE(3):S4=STORE(4):S5
     =STORE(5):S6=STORE(6):S7=STORE(
     7)
3210 RETURN
3999 REM CLEAR DATA SUB
4000 POSITION 2,16
4010 FOR Y=16 TO 19
4020 ? "{37 SPACES}"
4030 NEXT Y
4040 POSITION 0,20:? "{19 SPACES}"
4050 POSITION 0,21:? "{19 SPACES}"
4200 RETURN
4999 REM ENTER DATA INTO PROGRAM
5000 POKE 559,0
5010 GOSUB 3200
5020 GOSUB 5200
5030 ? 9000+N;" DATA ";S0;",";S1;","
     ;S2;",";S3;",";S4;",";S5;",";S6
     ;",";S7
5040 GOSUB 5210
5050 N=N+1
5060 POKE 764,255
5070 POKE 559,34
5080 GOTO 200
5200 ? CHR$(125):? :RETURN
```

```
5210 ? :? :? "CONT":POSITION Ø,Ø:POK
     E 842,13:STOP
5220 POKE 842,12:? CHR$(125):? :RETU
     RN
6000 ? "{10 SPACES}CHARACTER EDITOR"
6010 ? "{10 SPACES}{16 M}"
6020 RETURN
```

Program 2. Custom Characters

```
10 N=Ø
20 MEMTOP=PEEK(106)*256
30 CHBASE=MEMTOP-2048
40 REM CLEAR MEMORY FOR NEW CHARACTE
   R SET
50 FOR I=CHBASE TO CHBASE+1024
60 POKE I,Ø
70 NEXT I
80 REM POKE NEW CHARACTER SET INTO M
   EMORY
90 READ A
100 IF A=999 THEN 300:REM 999 IS END
     OF DATA FLAG
110 POKE CHBASE+N,A
120 N=N+1
130 GOTO 90
190 REM DATA STATEMENTS FOR SPACE,6
    CHARACTERS AND FLAG. FIRST CHARA
    CTER MUST BE A SPACE
195 REM LINE 220 IS A SPACE TO SKIP
    THE QUOTATION MARKS
200 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
210 DATA 32,33,35,35,35,35,255,255
220 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
230 DATA 112,112,112,112,248,248,248
    ,248
240 DATA 248,252,254,254,86,6,255,25
    5
250 DATA Ø,Ø,32,32,32,32,112,240
260 DATA 41,38,32,32,32,32,32,32
270 DATA Ø,Ø,Ø,Ø,Ø,32,32,48
280 DATA 999
290 REM SET GRAPHICS MODE
300 GRAPHICS 2
310 REM TELL COMPUTER WHERE TO FIND
    NEW CHARACTER SET
320 POKE 756,CHBASE/256
324 REM PRINT NEW CHARACTERS
325 POSITION 9,7
330 ? #6;"'%"
```

```
335 POSITION 9,8
340 ? #6;"&#"
345 POSITION 9,9
350 ? #6;"!$"
360 GOTO 360
```

Program 3. Fixed Playfield Demonstration

```
10 CLR
20 REM N = NUMBER OF CHARACTERS IN C
   HNEW$ STRING
30 N=31:CHORG=57344
40 REM DEFINE STRING
50 DIM CHNEW$(N)
60 CHNEW$="!#$%&'()*+,-./;<=>?@BGHJK
   MNPQVW"
70 REM FIND CHBASE
80 CHBASE=(PEEK(106)-8)*256
90 ? :? "  PLEASE WAIT, 760 NUMBERS
   TO MOVE"
100 REM COPY STANDARD CHARACTER SET
    FROM CHORG TO CHBASE
110 FOR I=0 TO 511
120 POKE CHBASE+I,PEEK(CHORG+I)
130 NEXT I
140 REM READ AND POKE CUSTOM DATA IN
    TO THE CHARACTERS IN STRING CHNE
    W$"
150 FOR I=1 TO N
160 CHADD=CHBASE+(ASC(CHNEW$(I))-32)
    *8
170 FOR J=0 TO 7
180 READ A
190 POKE CHADD+J,A
200 NEXT J
210 NEXT I
300 REM CUSTOM CHARACTER DATA
301 DATA 0,0,0,128,0,0,0,0
302 DATA 0,0,0,0,0,0,0,16
303 DATA 0,0,0,0,1,0,0,0
304 DATA 8,0,0,0,0,0,0,0
305 DATA 0,0,0,0,0,0,31,127
306 DATA 0,0,0,0,0,0,255,255
307 DATA 0,0,0,0,0,0,248,254
308 DATA 0,0,0,0,7,15,31,31
309 DATA 1,7,31,24,255,255,255,219
310 DATA 255,231,255,0,255,255,255,2
    19
311 DATA 128,224,248,24,255,255,255,
    219
```

```
312 DATA Ø,Ø,Ø,Ø,224,24Ø,248,248
313 DATA 31,31,15,7,1,1,3,2
314 DATA 219,255,255,255,127,16,32,6
    4
315 DATA 219,255,255,255,255,24,6Ø,6
    Ø
316 DATA 219,255,255,255,254,8,4,2
317 DATA 248,248,24Ø,224,128,128,192
    ,64
318 DATA 6,5,6,12,127,Ø,Ø,Ø
319 DATA 128,Ø,Ø,Ø,Ø,Ø,Ø,Ø
32Ø DATA 6Ø,126,126,126,6Ø,6Ø,6Ø,6Ø
321 DATA 1,Ø,Ø,Ø,Ø,Ø,Ø,Ø
322 DATA 96,16Ø,96,48,254,Ø,Ø,Ø
323 DATA 24,24,24,24,24,Ø,Ø,Ø
324 DATA 128,192,24Ø,24Ø,248,252,254
    ,255
325 DATA 129,195,231,255,255,255,255
    ,255
326 DATA 128,192,192,224,224,224,248
    ,255
327 DATA 1,3,7,31,63,63,127,255
328 DATA 1,3,7,7,15,31,63,255
329 DATA 255,255,255,255,255,254,249
    ,7
33Ø DATA 252,251,247,2Ø7,191,127,255
    ,255
331 DATA 255,255,255,255,255,255,255
    ,255
4ØØ REM PUT PLAYFIELD ON SCREEN
42Ø GRAPHICS 2:POKE 752,1
5ØØ REM TELL COMPUTER WHERE TO FIND
    NEW CHARACTER SET
51Ø POKE 756,CHBASE/256
53Ø SETCOLOR Ø,3,6:SETCOLOR 1,8,6:SE
    TCOLOR 2,1,1Ø:SETCOLOR 3,Ø,1Ø
54Ø REM PLOT CHARACTERS USING COLOR
    DATA
55Ø FOR ROW=Ø TO 9
56Ø FOR COLUMN=Ø TO 19
57Ø READ CHAR
58Ø COLOR CHAR
59Ø PLOT COLUMN,ROW
6ØØ NEXT COLUMN
61Ø NEXT ROW
62Ø REM PRINT STANDARD NUMBERS AND L
    ETTERS IN TEXT WINDOW
63Ø ? :? "FUEL:2568  STARDATE:174  A
    LTITUDE:39Ø";
```

```
640 REM BLINK ENGINE EXHAUST
650 FOR LUM=Ø TO 8 STEP 2
660 SETCOLOR Ø,3,LUM
670 NEXT LUM
680 GOTO 65Ø
699 REM CHARACTER COLOR DATA
700 DATA Ø,129,Ø,Ø,Ø,Ø,131,Ø,132,Ø,1
    33,Ø,Ø,132,Ø,Ø,Ø,Ø,Ø,131
701 DATA Ø,Ø,Ø,133,Ø,Ø,Ø,Ø,6,7,8,Ø,1
    29,Ø,Ø,129,Ø,133,Ø,Ø
702 DATA Ø,133,Ø,Ø,132,Ø,9,1Ø,11,11,
    11,12,13,129,Ø,Ø,131,Ø,129,Ø
703 DATA Ø,Ø,132,Ø,Ø,Ø,14,15,27,27,2
    7,28,29,Ø,133,Ø,Ø,Ø,Ø,132
704 DATA 2Ø2,133,Ø,132,Ø,133,3Ø,31,6
    4,64,64,98,1Ø3,Ø,129,Ø,2Ø6,2Ø2,1
    32,Ø
705 DATA 215,2Ø3,2Ø2,Ø,Ø,Ø,129,133,7
    2,72,72,Ø,133,Ø,2Ø6,2Ø3,215,215,
    2Ø5,Ø
706 DATA 215,215,215,2Ø5,133,Ø,2Ø6,2
    Ø3,215,2Ø2,133,Ø,Ø,2Ø8,215,215,2
    15,215,215,2Ø2
707 DATA 215,215,2Ø9,214,215,215,215
    ,215,215,215,215,2Ø9,214,215,215
    ,215,215,215,215,215
708 DATA 215,215,215,215,215,215,215
    ,215,215,215,215,215,215,215,215
    ,215,215,215,215,215
709 DATA 215,215,215,215,215,115,99,
    111,114,1Ø1,26,21,16,19,23,215,2
    15,215,215,215
```

3

The Four-Color Character Modes

Orson Scott Card

Here's how to unlock two "hidden" character modes on your Atari: ANTIC 4 and 5. Each character can display up to four colors at a time, and the effect can be exciting. This is a complete introduction to four-color character graphics, including subroutines that will help you use these modes in your own programs and a complete character set you can type in and use.

Two of the programs in this book are designed to help you make use of two "hidden" graphics modes on your Atari: the four-color character modes. "Four-Color Character Editor" allows you to create characters in these modes and save entire character sets to tape or disk. "Fontbyter" allows you to draw large or small screen displays using the character sets you created with the Four-Color Character Editor. To make full use of these utilities, it helps to know how the four-color character modes work.

ANTIC and the Character Modes

The GRAPHICS command automatically changes the way the ANTIC video chip in your Atari computer controls the television screen. GRAPHICS 0, 1, and 2 are text modes, putting characters like A, 7, or % on the screen; GRAPHICS 3 through 8 are pixel modes, in which you control the color of little squares, called pixels, on the screen.

The modes differ from each other in the size of each character or pixel. If you use a mode with larger pixels or characters, you will use up less memory to create your screen display—but you will also get fewer colors or poorer picture resolution.

When your program uses a character mode, the ANTIC processor scans through screen memory. Each byte of memory contains a number from 0 to 255. ANTIC uses that number as an index or pointer into character memory. The number 22 tells

ANTIC to skip 22 characters in order to find the one to display in that position on the screen.

Let's go through that process in detail.

The Display List

First, ANTIC checks locations 560 and 561 to get the address of the display list. When ANTIC jumps to the display list, it usually finds that the first three bytes each contain the number 112, which tells ANTIC to output several blank lines to the TV screen.

Screen memory address. ANTIC then finds an instruction that consists of the ANTIC mode number, or mode instruction, *plus* the number 64. The number 64 is a code that tells ANTIC to look for screen memory at the address contained in the next two bytes. Whenever ANTIC finds a 64 added to a mode instruction in the display list, it looks at the next two bytes to find the address of screen memory.

The first mode instruction *must* have a 64 and be followed by the screen memory address. If screen memory continues unbroken from there, you never need to give the address of screen memory again. Anytime you want to change screen memory, however, you only need to add 64 to the mode instruction for the line where you want the change to begin, and then give the new screen memory address in the next two bytes.

Mode instructions. The ANTIC mode number that is added to 64 is not the same as the graphics mode number you use with the GRAPHICS command. ANTIC modes range from 2 to 15. (Table 1 shows how the ANTIC mode numbers compare to graphics mode numbers.) ANTIC modes 2 through 7 are character modes; modes 8 through 15 are pixel modes.

There must be a mode instruction for every line on the screen. In ANTIC 2 (GRAPHICS 0) there are 24 lines on the screen—so there must be 24 mode instructions. In ANTIC 15 (GRAPHICS 8) there are 192 lines—and so you must have 192 mode instructions.

Close the display list. After the last mode instruction, there will be a 65—which is the 64 instruction added to 1. The 64 tells ANTIC to look for an address in the next two bytes, but the 1 tells it that it won't be screen memory, but rather the address of the start of the display list. ANTIC then waits until it's time to start displaying at the top-left corner of the TV screen again, then jumps back to the start of the display list and starts over.

Here's a short subroutine you can use in your own programs to create an ANTIC 4 or 5 display list. Lines 5-20 are not part of the subroutine—they're just there so you can RUN the program right now and see what ANTIC 4 and 5 do to the screen.

Atari BASIC will still let you type and enter program lines or instructions as if you were in GRAPHICS 0. That's fine if you're in ANTIC 4, since both GRAPHICS 0 and ANTIC 4 use 24 lines of 40 characters. But ANTIC 5 uses only 12 lines of 40 characters, so BASIC's screen handler will put half the screen display "below" the TV screen, out of sight.

Display List Maker

```
5 ? "What ANTIC mode do you want? (
  4 or 5"
10 TRAP 10:INPUT M:IF M<4 OR M>5 THE
   N M=2
15 ? "Writing a display list in ANTI
   C ";M:GOSUB 4000
20 END
4000 DL=PEEK(560)+256*PEEK(561):POKE
      DL+3,M+64
4005 FOR I=DL+6 TO DL+16+12*(M=4):PO
     KE I,M:NEXT I:POKE I,65:POKE I+
     1,0:POKE I+2,DL/256:RETURN
```

This routine works only if you are leaving the display list exactly where the latest GRAPHICS 0 statement left it.

Reading Screen Memory

After each mode instruction in the display list, ANTIC goes to screen memory and reads the next line's worth of information. If a line is 40 units wide, ANTIC reads 40 bytes. In a pixel mode, ANTIC interprets those bytes directly as instructions telling it what color to use for the squares on the screen. In a character mode, however, ANTIC uses the bytes as an index into the character set.

Finding the character set. ANTIC finds the character set by using the number held in location 756. When the computer powers up, that number points to the page number (or high byte) of the address of the built-in character set. You can change that number to point to your own character set. Make sure that your character set begins on a 1K boundary, however. The easiest way to make sure that the address you are POKEing into location 756 is on a 1K boundary is to use the following routine. The variable CHBAS is the *high byte* of the starting address of your character set. TOP is where you are telling the computer the top of memory is. Anything you put above TOP will be left alone by the computer.

```
500 TOP=PEEK(106)-8
510 POKE 106,TOP
520 CHBAS=TOP+4
```

This fools the computer into thinking that usable memory ends eight pages (2K) sooner than it really does. Now you have plenty of space that the computer won't touch. The character set itself needs only four pages (1K), but the other four pages will hold your display list and other safe memory. In fact, if you subtract more pages, you can put your screen memory in this safe area, too.

```
530 CH=CHBAS*256
```

If you multiply CHBAS by 256, you get CH, the full address of the character set, rather than just the page number, or high byte, of its address.

```
540 OPEN #1,4,0,"D:CUSTOM.SET":FOR I
    =0 TO 1024:GET #1,N:POKE CH+I,N:
    NEXT I: CLOSE
```

This line opens the disk file containing your character set ("D:CUSTOM.SET") and loads it (slowly) into screen memory.

```
550 POKE 756,CHBAS
```

From the moment you make this POKE, ANTIC will use your character set instead of the built-in character set.

Fast loading. Here is a subroutine you can add to your own programs. It loads your character set from disk in a few seconds, using a machine language routine, and then writes a display list. Most of the time you will completely eliminate lines 5 and 6, the PRINT ("?") statements in lines 4000 and 4020, and all of lines 4025 and 4030; instead, your program will simply assign your character set's disk filename to the variable CHSET$ and the ANTIC mode number to the variable *M*. The excess material is included here so that you can try out the routine and see how it works.

Character Set Loader

```
5 DIM CHSET$(20):CHSET$="D:CASTLE.SE
  T":GOSUB 4000
6 END
```

```
4000 A=PEEK(106):TOP=A-8:CHBAS=TOP+4
     :DL=256*TOP:POKE 106,TOP:CH=CHB
     AS*256:GRAPHICS 0:? "Loading ";
     CHSET$
4005 X=16:ICCOM=834:ICBADR=836:ICBLE
     N=840:SL=PEEK(88):SH=PEEK(89)
4010 OPEN #1,4,0,CHSET$
4015 POKE ICBADR+X+1,CHBAS:POKE ICBA
     DR+X,0:POKE ICBLEN+X+1,4:POKE I
     CBLEN+X,0
4020 POKE ICCOM+X,7:I=USR(ADR("hhh▣L
     V◘"),X):CLOSE #1:? "What ANTIC
     mode do you want? (4 or 5)"
4025 TRAP 4025:INPUT M:IF M<4 OR M>5
      THEN 4025
4030 ? "Writing the display list for
      ANTIC ";M
4035 FOR I=0 TO 2:POKE DL+I,112:NEXT
      I:POKE DL+3,M+64:POKE DL+4,SL:
     POKE DL+5,SH
4040 FOR I=DL+6 TO DL+16+12*(M=4):PO
     KE I,M:NEXT I:POKE I,65:POKE I+
     1,0:POKE I+2,DL/256
4045 POKE 756,CHBAS:POKE 560,0:POKE
     561,DL/256
```

Reading the character set. When ANTIC reads a byte of screen memory, it uses it as an index into the character set. Let's say that ANTIC found a 34 in screen memory. Since each character in the set uses up eight bytes for its definition, ANTIC will look for the character at CH + 8*34, or 272 bytes after the start of the character set. ANTIC reads the eight bytes of that character pattern; it will display the character pattern in the next location on the screen. Then it goes to screen memory, reads the next byte, finds that character pattern in the character set, and so on.

ANTIC 4 and 5 Character Modes

ANTIC 4 handles screen memory exactly like GRAPHICS 0 (ANTIC 2), the normal text mode on the Atari. There are 24 lines of 40 characters each on the screen. However, the *characters* are interpreted very differently.

Normal character patterns. No matter what character mode you are using, the characters are created following the pattern formed by eight bytes. Each byte consists of eight bits. Think of the bytes as if they were stacked on top of each other, making a square eight bits wide and eight bytes high, as in Table 2.

ANTIC 2 (GRAPHICS 0) reads the character pattern in a straightforward way. If a bit in the character pattern is set to 1, it is *on*, and one dot of the character's color is displayed; if the bit is set to 0, the background color is displayed in that spot. The pattern for a letter *A* might look like the one shown in Table 3.

Four-color character patterns. ANTIC 4 and 5 cannot use exactly this system, since the same eight bytes must tell ANTIC not only which bits are on and which are off, but also which color the dot on the screen should be. One bit just won't do. So the four-color character modes treat the bits as pairs, so that *two* bits control each dot on the screen, as shown in Table 4.

However, this would make each character only half as wide, and it would take 80 characters to fill a line. To avoid this, each bit-pair in ANTIC 4 characters controls the color of *two* dots on the screen. Now the characters are just as wide as the characters in ANTIC 2 (GRAPHICS 0), but the horizontal resolution is only half as good.

That is why, in these four-color character modes, it is nearly impossible to create letters like *W* and *M*, and very hard to distinguish between *N* and *H*. The four-color modes are not particularly good for alphanumeric characters. However, they are wonderful for making the building blocks of elaborate full-color drawings.

The five colors. Each bit-pair has four possible combinations: 00, 01, 10, and 11. These correspond to the decimal numbers 0, 1, 2, and 3. A bit-pair with the value of 00 will cause the background color, stored at memory location 712, to be displayed. A bit-pair with the value of 01 will display the color stored at location 708; the bit-pair 10 (decimal 2) will display the color at location 709; and the bit-pair 11 (decimal 3) will display the color stored at location 710.

There is a fifth color available. If a character is entered into screen memory in *inverse* mode (the character number *plus* 128), then any bit-pair 11 (decimal 3) in that character will display, not the color at 710, but the color at 711. This means that if you plan your character set carefully, you can have five colors on your screen at one time.

We still call them four-color character modes, however, because no one character can display more than four colors at a time.

Table 5 shows a simple shape, an apple tree with fruit on it. The trunk is brown, the leaves are green, the fruit is red, and the background is black. However, this pattern creates only *half* the tree—it is assumed that the other half is held in another character, and the two would be combined to create the full tree.

Also, the fruit is created with the bit-pair 11 (decimal 3). This allows us to enter the same character in inverse mode and display different-colored fruit—orange, for example, instead of red. Or fruit could ripen gradually from green to bright red. Careful planning can result in a great deal of freedom in creating your screen displays.

How to Use the Utilities

The Four-Color Character Editor will let you create your own ANTIC 4 and 5 character sets. This will be a time-consuming project, but once you have a character set made, you can use it again and again to make many different drawings.

Fontbyter will let you use your own character sets—or the Castle Maker character set included here—to make your own drawings and save them, perhaps to use them in programs. You can make drawings many times the size of the TV screen and scroll through them.

You don't have to have a particular programming goal in mind. Though both programs are long and will take quite a bit of time to type in, they can be used over and over again. You'll have a hard time deciding whether you're programming, creating artwork, or just playing. The truth is, with ANTIC 4 and 5, you can do all three at once.

Table 1. Character Modes

Character Modes		Characters or pixels per line	Comments
GRAPHICS	ANTIC		
0	2	40	The default mode of the Atari. The border and background can be different colors, but the characters are the same color as the background—only the brightness is different. Twenty-four lines on the screen.
—	3	40	Rarely used. It is identical to GRAPHICS 0 except that the top two lines of each lowercase character are put on the bottom. It can be used to create true descenders on lowercase letters.
12*	4	40	Identical to GRAPHICS 0 in the way it uses screen memory and the size of the character set (all 256 characters are available), but four colors are possible in *each* character, and by using inverse characters, five colors can be displayed on the screen at once. However, the horizontal resolution is only half as good as in GRAPHICS 0. Twenty-four lines on the screen.
13*	5	40	Identical to ANTIC 4 except that each character is twice as tall. This gives the characters a stretched-out look, but fills up the screen while using half as much memory. Twelve lines on the screen.
1	6	20	Uses only 64 characters, but each character can be displayed with *one* of four different colors. Twenty- four lines per screen.
2	7	20	The same as GRAPHICS 1 (ANTIC 6) except the characters are twice as tall. Twelve lines per screen.

*XL Models

Table 2. The 8 x 8 Character Matrix

Bits	7	6	5	4	3	2	1	0
Byte 1	0	0	0	0	0	0	0	0
Byte 2	0	0	0	0	0	0	0	0
Byte 3	0	0	0	0	0	0	0	0
Byte 4	0	0	0	0	0	0	0	0
Byte 5	0	0	0	0	0	0	0	0
Byte 6	0	0	0	0	0	0	0	0
Byte 7	0	0	0	0	0	0	0	0
Byte 8	0	0	0	0	0	0	0	0

Table 3. A GRAPHICS 0 Character Pattern

Bits	7	6	5	4	3	2	1	0
Byte 1	0	0	0	0	0	0	0	0
Byte 2	0	0	0	1	1	0	0	0
Byte 3	0	0	1	1	1	1	0	0
Byte 4	0	1	1	0	0	1	1	0
Byte 5	0	1	1	1	1	1	1	0
Byte 6	0	1	1	0	0	1	1	0
Byte 7	0	1	1	0	0	1	1	0
Byte 8	0	0	0	0	0	0	0	0

```

      1 1
    1 1 1 1
  1 1     1 1
  1 1 1 1 1 1
  1 1     1 1
  1 1     1 1

```

Table 4. The 4 x 8 ANTIC 4 and 5 Matrix

Bits	7-6	5-4	3-2	1-0
Byte 1	00	00	00	00
Byte 2	00	00	00	00
Byte 3	00	00	00	00
Byte 4	00	00	00	00
Byte 5	00	00	00	00
Byte 6	00	00	00	00
Byte 7	00	00	00	00
Byte 8	00	00	00	00

Table 5. An ANTIC 4 Character Pattern

Bits	Bit Patterns*				"On" Bits				Colors Displayed			
	7-6	5-4	3-2	1-0								
Byte 1	00	10	00	10		10		10		GRE		GRE
Byte 2	10	11	10	11	10	11	10	11	GRE	RED	GRE	RED
Byte 3	10	10	10	10	10	10	10	10	GRE	GRE	GRE	GRE
Byte 4	00	10	11	10		10	11	10		GRE	RED	GRE
Byte 5	00	10	10	10		10	10	10		GRE	GRE	GRE
Byte 6	00	00	00	01				01				BRN
Byte 7	00	00	00	01				01				BRN
Byte 8	00	00	00	01				01				BRN

*00 = Color Register 4 (background, memory location 712)
01 = Color Register 0 (memory location 708)
10 = Color Register 1 (memory location 709)
11 = Color Register 2 (memory location 710)
11 *inverse* = Color Register 3 (memory location 711)

Castle Maker—A Character Set

```
900 OPEN #1,8,Ø,"D1:CASTLE.SET"
91Ø FOR I=1 TO 1Ø24:READ N:PUT #1,N:
    NEXT I:CLOSE #1:? I:END
1ØØØ DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1ØØ8 DATA 255,84,84,84,84,84,84,68
1Ø16 DATA 255,21,21,21,21,21,21,17
1Ø24 DATA 191,191,186,255,171,171,17
     1,255
1Ø32 DATA Ø,25,Ø,Ø,Ø,Ø,Ø,Ø
1Ø4Ø DATA Ø,4,16,64,85,64,16,4
1Ø48 DATA 1,3,1,3,1,3,1,3
1Ø56 DATA 64,192,64,192,64,192,64,19
     2
1Ø64 DATA Ø,Ø,3,15,63,252,24Ø,192
1Ø72 DATA Ø,Ø,192,24Ø,252,63,15,3
1Ø8Ø DATA 255,3,3,3,3,3,3,3
1Ø88 DATA 255,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1Ø96 DATA Ø,Ø,Ø,Ø,68,32,Ø,Ø
11Ø4 DATA 25Ø,25Ø,255,255,175,175,17
     Ø,255
1112 DATA Ø,8,17,Ø,Ø,Ø,Ø,Ø
112Ø DATA 17,8,Ø,Ø,Ø,Ø,Ø,Ø
1128 DATA 252,63,15,3,Ø,Ø,Ø,Ø
1136 DATA 85,85,85,85,85,85,85,85
1144 DATA 255,255,3,3,12,24Ø,Ø,Ø
1152 DATA 255,255,Ø,Ø,Ø,Ø,Ø,Ø
```

```
1160 DATA 255,255,192,192,48,15,0,0
1168 DATA 0,0,0,0,0,0,192,240
1176 DATA 0,0,0,0,0,0,3,15
1184 DATA 3,3,3,12,12,240,0,0
1192 DATA 0,0,3,15,63,252,243,195
1200 DATA 63,252,240,192,0,0,0,0
1208 DATA 42,4,4,4,4,0,0,0
1216 DATA 255,192,192,192,192,192,19
     2,192
1224 DATA 0,0,192,240,252,63,207,195
1232 DATA 171,171,171,171,251,251,25
     5,255
1240 DATA 192,192,192,48,48,15,0,0
1248 DATA 255,255,255,255,171,171,17
     1,255
1256 DATA 0,0,0,0,0,32,68,0
1264 DATA 186,186,186,255,171,171,17
     1,255
1272 DATA 186,255,171,255,186,255,17
     1,255
1280 DATA 0,0,0,0,255,195,195,195
1288 DATA 0,0,0,0,0,0,0,255
1296 DATA 64,80,84,85,85,85,85,85
1304 DATA 3,3,3,3,3,3,3,255
1312 DATA 192,192,192,192,192,192,19
     2,192
1320 DATA 3,3,3,3,3,3,3,3
1328 DATA 64,64,80,80,84,84,85,85
1336 DATA 255,255,255,255,255,255,25
     5,255
1344 DATA 0,0,0,0,171,171,171,255
1352 DATA 171,171,171,255,171,171,17
     1,255
1360 DATA 192,192,224,224,232,232,23
     4,255
1368 DATA 3,3,11,11,43,43,171,255
1376 DATA 170,170,170,170,175,175,25
     5,255
1384 DATA 175,175,255,255,251,251,17
     1,255
1392 DATA 0,0,0,0,1,5,21,85
1400 DATA 0,0,0,0,64,80,84,85
1408 DATA 192,192,192,192,192,192,19
     2,255
1416 DATA 87,23,87,3,3,3,3,3
1424 DATA 1,1,5,5,21,21,85,85
1432 DATA 0,0,0,255,186,255,171,255
1440 DATA 1,5,21,85,85,85,85,85
1448 DATA 255,195,195,195,255,0,0,0
```

```
1456 DATA 213,212,213,192,192,192,19
     2,192
1464 DATA 255,195,195,195,255,195,19
     5,195
1472 DATA 171,171,43,43,11,11,3,3
148Ø DATA 11,11,11,15,3,3,3,3
1488 DATA 234,234,232,232,224,224,19
     2,192
1496 DATA 224,224,224,24Ø,192,192,19
     2,192
15Ø4 DATA 186,186,186,255,43,43,43,6
     3
1512 DATA 85,2Ø,16,4,1,17,68,16
152Ø DATA 2,2,1Ø,22,22,86,86,85
1528 DATA 252,236,172,44,5,5,5,245
1536 DATA 84,8Ø,64,16,68,68,16,64
1544 DATA 4,4,21,2Ø,85,85,84,8Ø
1552 DATA 4,5,1,21,31,31,31,95
156Ø DATA Ø,Ø,64,84,84,Ø,Ø,Ø
1568 DATA Ø,Ø,1,21,21,Ø,Ø,Ø
1576 DATA 16,16,84,2Ø,85,85,21,5
1584 DATA Ø,3,3,15,61,4,17,32
1592 DATA 85,133,129,149,154,154,154
     ,154
16ØØ DATA 128,128,16Ø,148,148,149,14
     9,85
16Ø8 DATA 85,16,4,1,17,68,16,64
1616 DATA 85,8Ø,64,Ø,Ø,64,64,16
1624 DATA 21,5,1,4,17,17,4,1
1632 DATA 64,24Ø,176,188,95,68,81,66
164Ø DATA 66,81,93,93,93,93,253,85
1648 DATA 85,85,85,85,85,85,51,51
1656 DATA 32,17,29,29,29,29,31,21
1664 DATA 85,82,66,86,166,166,166,16
     6
1672 DATA 66,81,221,221,221,221,253,
     85
168Ø DATA Ø,3,3,3,1,4,16,16
1688 DATA 63,59,58,56,8Ø,8Ø,8Ø,95
1696 DATA 16,8Ø,64,84,244,244,244,24
     5
17Ø4 DATA 85,5,1,Ø,Ø,1,1,4
1712 DATA 64,24Ø,176,176,8Ø,68,65,65
172Ø DATA 85,2Ø,4,16,64,68,17,4
1728 DATA 84,68,84,16,16,2Ø,16,2Ø
1736 DATA 4,16,64,Ø,8Ø,68,17,4
1744 DATA 255,195,255,12,12,252,6Ø,2
     52
1752 DATA 5,7,12,48,63,53,53,63
```

```
1760 DATA 192,0,0,0,252,92,92,252
1768 DATA 0,0,20,125,255,195,195,255
1776 DATA 0,0,0,5,21,85,84,85
1784 DATA 0,0,5,16,64,64,80,5
1792 DATA 0,0,0,0,0,21,65,16
1800 DATA 0,0,80,4,1,1,5,80
1808 DATA 68,100,100,32,32,20,33,32
1816 DATA 0,0,0,0,64,21,85,81
1824 DATA 0,0,16,4,1,65,84,0
1832 DATA 16,16,16,68,68,65,17,81
1840 DATA 60,56,56,21,80,80,20,48
1848 DATA 16,16,16,20,17,17,64,69
1856 DATA 60,28,223,215,215,215,215,
     235
1864 DATA 48,48,240,240,252,255,252,
     65
1872 DATA 48,48,60,63,63,63,63,24
1880 DATA 48,60,60,204,204,204,204,6
     8
1888 DATA 48,236,239,16,100,100,85,4
     8
1896 DATA 60,248,248,32,96,84,80,80
1904 DATA 16,4,0,85,5,0,4,16
1912 DATA 0,32,32,32,32,32,21,32
1920 DATA 16,80,69,85,64,80,68,17
1928 DATA 60,40,40,21,84,84,84,48
1936 DATA 240,176,176,16,20,20,80,48
1944 DATA 0,0,0,0,1,84,85,69
1952 DATA 1,4,4,84,16,16,64,64
1960 DATA 0,4,1,1,81,85,69,21
1968 DATA 60,44,44,15,19,80,16,16
1976 DATA 4,20,68,21,85,1,1,5
1984 DATA 250,250,186,255,171,171,17
     1,255
1992 DATA 186,186,186,255,232,232,23
     2,252
2000 DATA 0,0,0,0,0,100,0,0
2008 DATA 255,60,60,255,60,48,48,0
2016 DATA 3,12,48,192,80,196,1,0
```

3

Four-Color Character Editor

Tim Kilby

Creating multicolor character sets is no longer tedious guesswork. With this editor you can see the characters as you create them, with the exact colors you want.

Two of the most effective graphics modes on the Atari are hidden away where you can't access them from BASIC. ANTIC modes 4 and 5 require you to use a custom display list—and then you can't do anything until you develop your own character sets. But it's worth the effort.

What makes these two modes so valuable? For one thing, each character can display up to four colors, and when you enter characters in inverse mode, any pixels in Color 3 are displayed instead as a fifth color. For another thing, you can get better resolution than GRAPHICS 7, while using up only a fraction of the memory. This lets you create easy-to-scroll displays that are many times larger than is possible in the pixel modes.

With the right tools, ANTIC 4 and 5 are as easy to use as any other. This is one of those tools.

What You See

After a few moments of initialization, the editing workspace appears on the screen.

The top third of the screen is the Character Grid. Actually, two grids are shown with little dots on a black background. The left-hand grid shows the current character roughly as it appears in ANTIC 4, but much enlarged. The right-hand grid shows the character bit-pattern. In both grids, the rectangles are the correct color. This means you don't have to keep track of which bits should be on or off in order to access Colors 1, 2, or 3.

On the left-hand grid there is also a red rectangle. This is your cursor. It moves as you command it, and you can change the color of whatever rectangle it rests upon.

The Message Field, below the grids, is light blue. At the beginning it shows the Menu, which tells you which commands are available to you.

The Character Field displays the entire character set in ATASCII numerical order.

And the black Display Field shows the character currently in the grid in its actual size, in regular and inverse modes, in ANTIC 4 (short) and ANTIC 5 (tall), singly and clustered.

How to Use the Editor

Load. Press *L* to load a previously saved character set. The program assumes that the character set will be on disk drive 1 and will have the extender ".SET". Therefore, you need to enter only the eight-letter filename. As soon as the character set is loaded, it will be displayed in the Character Field.

Save. Press *S* to save the character set as it is now displayed in the Character Field. Again, you need to enter only the eight-letter filename. To change the default device from "D1:" to "D*n*:" or "C:", just change line 820.

1, 2, and 3. Press 1 to select Color 1; press 2 to select Color 2; press 3 to select Color 3. Whichever color was last selected will be placed in the character at the cursor position when you press the joystick button.

Edit. Press *E* to select a new character to edit. The Message Field will display the prompt "- Select character -". The character you select will then be displayed in the Character Grid and in the Display Field.

Menu. Press *M* or the space bar to display the Menu in the Message Grid.

Rub. Press *R* to erase completely the character now in the Character Grid. It will immediately become a blank.

Copy. Press C to copy one character's pattern into another's place in the character set. Whatever character is then being edited will immediately be replaced by the copy-character's pattern—both characters will then be identical. This command lets you move characters around. Remember, though, that the character you copy *to* will disappear—it's a good idea, if you don't want to lose it, to copy that character to another position first. If you wanted to trade the positions of *B* and *A*, you would first choose to edit the blank character (space bar). Press C and then *B*; the blank character will then be replaced by *B*. Then choose to edit *B*, and copy from *A*. Then choose to edit *A* and copy from the blank character. Finally, choose to edit the blank character again, and type *R* to erase it—it will be blank again, and *A* and *B* will have traded places.

Test. Press *T* to enter Test Mode. In this mode, the Display Field goes blank, and you can test your characters simply by typing them normally at the keyboard. The cursor control keys retain their normal function; to print the ARROW key characters, you must press ESC first, as in GRAPHICS 0. You can still change colors (see Joystick 2, below). Pressing the INVERSE key will toggle you back and forth between inverse and regular mode. Exit Test Mode by pressing RETURN. No other editing commands function in Test Mode.

SELECT. Press SELECT to toggle the Character Field between ANTIC 4 and GRAPHICS 0.

Joystick 1. Joystick 1 controls the cursor's movement in the Character Grid. Pressing the button on Joystick 1 causes the cursor rectangle to toggle on or off. If the cursor is over a blank rectangle, it will then display the currently selected color. If the cursor is over a colored rectangle, it will then become blank.

Joystick 2. Joystick 2 controls the actual color values displayed by Colors 1, 2, and 3, and the background and inverse colors as well. Moving the joystick toward you and away from you controls Color 1. Moving the joystick left and right controls Color 2. Moving the joystick toward you and away from you with the OPTION button pressed controls the inverse color. Moving the joystick toward you and away from you with the joystick button pressed controls Color 3. And moving the joystick left and right with the joystick button pressed controls the background color. This lets you see how different color combinations work with the characters you have created. You can also change colors during Test Mode. (GRAPHICS 0 reads the color registers differently, so the Message Field and Character Field can behave oddly during color changes.)

To quit editing, press RESET.

Four-Color Character Editor

```
20 REM  (C) COPYRIGHT 1982 TIM KILBY
   ■
30 DIM A$(1),O$(1),CLEAR2$(1),ZERO$(
   1),CLEAR1$(1),CLEAR4$(1),FILE$(15
   ),NAME$(15),A(7,8)
40 GOTO 1180
```

```
50 SOUND Ø,Y/2+1ØØ-X/4,1Ø,4:FOR D=1
   TO 8:NEXT D:SOUND Ø,Ø,Ø,Ø:RETURN
6Ø REM  MENU
7Ø GOSUB 11Ø:POSITION 1,Ø
8Ø ? "Edit{5 SPACES}Copy{5 SPACES}Lo
   ad Set{4 SPACES}Rub"
9Ø ? "Color 1,2,or 3{4 SPACES}Save S
   et{4 SPACES}Test"
1ØØ ? "SELECT {ESC}{DOWN}mode{ESC}
    {DOWN}{7 SPACES}* CHOOSE ONE *";
    :RETURN
11Ø CLEAR2$=ZERO$(1,12Ø):RETURN
12Ø REM  SCREEN MEMORY ORIENTATION
13Ø POKE 87,3:POKE 88,PEEK(DL+4):POK
    E 89,PEEK(DL+5):RETURN
14Ø POKE 82,1:POKE 87,Ø:POKE 88,HOME
    +1ØØ-INT((HOME+1ØØ)/256)*256:POK
    E 89,INT((HOME+1ØØ)/256):RETURN
15Ø POKE 82,4:POKE 87,Ø:POKE 88,HOME
    +22Ø-INT((HOME+22Ø)/256)*256:POK
    E 89,INT((HOME+22Ø)/256):RETURN
16Ø POKE 82,Ø:POKE 87,Ø:POKE 88,HOME
    +38Ø-INT((HOME+38Ø)/256)*256:POK
    E 89,INT((HOME+38Ø)/256):RETURN
17Ø POKE 82,Ø:POKE 87,Ø:POKE 88,HOME
    +46Ø-INT((HOME+46Ø)/256)*256:POK
    E 89,INT((HOME+46Ø)/256):RETURN
18Ø GOSUB 16Ø:POKE 766,1:POSITION 11
    ,1:? CHR$(CHR):POSITION 18,1:? C
    HR$(CHR+128):POSITION 25,1
19Ø FOR A=1 TO 5:? CHR$(CHR);:NEXT A
    :? :GOSUB 17Ø:POSITION 11,1:? CH
    R$(CHR):POSITION 18,1:? CHR$(CHR
    +128)
2ØØ POSITION 25,1:FOR A=Ø TO 5:? CHR
    $(CHR);:NEXT A:POKE 766,Ø:RETURN
21Ø REM  PLOT POINTS
22Ø GOSUB 13Ø:F=X/4:G=(Y-2Ø)/4:LOCAT
    E 8+F,G+1,A:SOUND Ø,2Ø+G-F,1Ø,2:
    COLOR C*(A=Ø):PLOT 8+F,G+1:PLOT
    9+F,G+1
23Ø IF C=1 THEN PLOT 28+F,G+1:COLOR
    Ø:PLOT 27+F,G+1:A(G,F+1)=INT(2^(
    6-F)+Ø.1)
24Ø IF C=2 THEN PLOT 27+F,G+1:COLOR
    Ø:PLOT 28+F,G+1:A(G,F)=INT(2^(7-
    F)+Ø.1)
```

```
250 IF C=3 THEN PLOT 27+F,G+1:PLOT 2
    8+F,G+1:A(G,F)=INT(2^(7-F)+Ø.1):
    A(G,F+1)=INT(2^(6-F)+Ø.1)
260 IF A>Ø THEN A(G,F)=Ø:A(G,F+1)=Ø
270 A(G,8)=Ø:FOR D=Ø TO 7:A(G,8)=A(G
    ,8)+A(G,D):NEXT D:POKE CHBASE+CH
    R*8+G,A(G,8)
280 SOUND Ø,Ø,Ø,Ø:GOSUB 14Ø:RETURN
290 REM  EDIT 
300 GOSUB 11Ø:GOSUB 34Ø
310 POSITION 1Ø,1:? "- Select charac
    ter -";
320 GET #3,CHR:GOSUB 36Ø:GOSUB 14Ø:R
    ETURN
330 FOR A=Ø TO 7:POKE CHBASE+CHR*8+A
    ,Ø:NEXT A
340 CLEAR1$=ZERO$(1,1ØØ)
350 FOR A=Ø TO 7:FOR B=Ø TO 8:A(A,B)
    =Ø:NEXT B:NEXT A:RETURN
360 REM  PLOT CHARACTER 
370 POSITION 6,Ø:? "Use joystick to
    move cursor.":POSITION 8,1:? "Pr
    ess FIRE to plot point."
380 POSITION 14,2:? "(M for MENU)";
390 GOSUB 18Ø
400 GOSUB 13Ø:IF CHR>127 THEN CHR=CH
    R-128
410 IF CHR>127 THEN CHR=CHR-128
420 IF CHR>31 AND CHR<96 THEN CHR=CH
    R-32:GOTO 44Ø
430 IF CHR<32 THEN CHR=CHR+64
440 R=CHBASE+CHR*8:FOR A=Ø TO 7:D=PE
    EK(R+A):B=A+1
450 F=Ø:IF D>127 THEN D=D-128:F=F+1:
    COLOR 2:PLOT 27,B:PLOT 8,B:PLOT
    9,B:A(A,Ø)=128:A(A,8)=A(A,8)+A(A
    ,Ø)
460 IF D>63 THEN D=D-64:F=F+1:COLOR
    1:PLOT 28,B:PLOT 8,B:PLOT 9,B:A(
    A,1)=64:A(A,8)=A(A,8)+A(A,1)
470 IF F>1 THEN COLOR 3:PLOT 27,B:PL
    OT 28,B:PLOT 8,B:PLOT 9,B
480 F=Ø:IF D>31 THEN D=D-32:F=F+1:CO
    LOR 2:PLOT 29,B:PLOT 1Ø,B:PLOT 1
    1,B:A(A,2)=32:A(A,8)=A(A,8)+A(A,
    2)
490 IF D>15 THEN D=D-16:F=F+1:COLOR
    1:PLOT 3Ø,B:PLOT 1Ø,B:PLOT 11,B:
    A(A,3)=16:A(A,8)=A(A,8)+A(A,3)
```

```
500 IF F>1 THEN COLOR 3:PLOT 29,B:PL
    OT 30,B:PLOT 10,B:PLOT 11,B
510 F=0:IF D>7 THEN D=D-8:F=F+1:COLO
    R 2:PLOT 31,B:PLOT 12,B:PLOT 13,
    B:A(A,4)=8:A(A,8)=A(A,8)+A(A,4)
520 IF D>3 THEN D=D-4:F=F+1:COLOR 1:
    PLOT 32,B:PLOT 12,B:PLOT 13,B:A(
    A,5)=4:A(A,8)=A(A,8)+A(A,5)
530 IF F>1 THEN COLOR 3:PLOT 31,B:PL
    OT 32,B:PLOT 12,B:PLOT 13,B
540 F=0:IF D>1 THEN D=D-2:F=F+1:COLO
    R 2:PLOT 33,B:PLOT 14,B:PLOT 15,
    B:A(A,6)=2:A(A,8)=A(A,8)+A(A,6)
550 IF D>0 THEN F=F+1:COLOR 1:PLOT 3
    4,B:PLOT 14,B:PLOT 15,B:A(A,7)=1
    :A(A,8)=A(A,8)+A(A,7)
560 IF F>1 THEN COLOR 3:PLOT 33,B:PL
    OT 34,B:PLOT 14,B:PLOT 15,B
570 NEXT A:GOSUB 140:RETURN
580 REM  TEST CHARACTERS 
590 GOSUB 110:POSITION 5,0:? "Your t
    yped characters":POSITION 7,1:?
    "will appear below.{5 SPACES}NOR
    MAL"
600 POSITION 2,2:? "- Press  RETURN 
     for menu -";
610 D=1:POKE 764,255:CLEAR4$=ZERO$:G
    OSUB 340:GOSUB 160:POKE 84,0:POK
    E 85,0:POKE 53259,1:A$=ZERO$(1,1
    27)
620 POKE 53251,36:A$(86,89)="{4 C}"
630 FOR D=0 TO 5:POKE DL+D+23,4:NEXT
     D:POKE DL+29,65:POKE DL+30,PEEK
    (560):POKE DL+31,PEEK(561)
640 A=PEEK(633):ON A<15 GOSUB 1500:I
    F PEEK(764)=255 THEN 640
650 IF PEEK(764)=39 OR PEEK(764)=103
     THEN A=PEEK(84):B=PEEK(85):GOTO
     710
660 GET #3,CHR:IF CHR=155 THEN GOSUB
     340:CLEAR4$=ZERO$:A$=ZERO$(1,12
    7):GOSUB 140:POKE 694,0:POKE 702
    ,64:GOTO 750
670 ? CHR$(CHR);:IF PEEK(84)<0 THEN
    POKE 84,5
675 IF PEEK(85)>38 THEN POKE 85,0
680 IF PEEK(84)>5 THEN POKE 84,0
690 POKE 53251,PEEK(85)*4+36:A=PEEK(
    84)*4+86:A$=ZERO$(1,127):A$(A,A+
    3)="{4 C}"
```

```
700 POKE 764,255:GOTO 640
710 POKE 694,128*(PEEK(694)=0):POKE
    53279,0:GOSUB 140
720 IF PEEK(694)=128 THEN POSITION 2
    9,1:? " INVERSE "
730 IF PEEK(694)=0 THEN POSITION 29,
    1:? " NORMAL  "
740 POKE 764,255:GOSUB 160:POKE 84,A
    :POKE 85,B:GOTO 640
750 GOSUB 760:GOSUB 140:POKE 53259,3
    :RETURN
760 POKE DL+23,4:POKE DL+24,4:POKE D
    L+25,5:POKE DL+26,5:POKE DL+27,6
    5:POKE DL+28,PEEK(560):POKE DL+2
    9,PEEK(561)
770 RETURN
780 REM  LOAD CHARACTER SET 
790 FILE$="load":GOSUB 800:TRAP 890:
    OPEN #1,4,0,FILE$:POKE 850,7:GOS
    UB 870:RETURN
800 POKE 752,0:GOSUB 140:GOSUB 110:P
    OSITION 2,0:? "Enter name to ";F
    ILE$;". (1-8 characters)"
810 POSITION 16,1:INPUT NAME$:POKE 7
    52,1:IF NAME$="" THEN POP :RETUR
    N
811 FOR A=1 TO LEN(NAME$):IF NAME$(A
    ,A)=":" THEN NAME$=NAME$(A+1,LEN
    (NAME$)):POP :GOTO 811
812 IF NAME$(A,A)="." THEN NAME$=NAM
    E$(1,A-1):POP :GOTO 811
813 IF ASC(NAME$(A,A))<65 OR ASC(NAM
    E$(A,A))>90 THEN POP :POP :RETUR
    N
814 NEXT A
820 FILE$="D:":REM * Cassette users
    {3 SPACES}should replace "D:" wi
    th "C:".  The{3 SPACES}remaining
     file name is irrelevant.
830 FILE$(LEN(FILE$)+1)=NAME$:FILE$(
    LEN(FILE$)+1)=".SET"
840 GOSUB 110:RETURN
850 REM  SAVE CHARACTER SET 
860 FILE$="save":GOSUB 800:TRAP 890:
    OPEN #1,8,0,FILE$:POKE 850,11:GO
    SUB 870:RETURN
870 POKE 852,0:POKE 853,CHBASE/256:P
    OKE 856,0:POKE 857,4:POKE 756,CH
    BASE/256:A=USR(1555)
```

```
880 CLOSE #1:TRAP 32767:POKE 54286,1
    92:POKE 756,224:RETURN
890 GOSUB 110:POSITION 1,0:? CHR$(25
    3);"An ERROR ";PEEK(195);" has o
    ccurred."
900 IF PEEK(195)=165 THEN ? "Imprope
    r file name - try again."
910 IF PEEK(195)=170 THEN ? "File no
    t found - try again."
920 ? "Press any key to continue.";:
    GET #3,KEY:GOTO 880
930 A=STICK(1):IF A<>15 THEN GOSUB 1
    500:REM  JOYSTICK 
940 A=STICK(0):B=STRIG(0)
950 IF A=7 THEN X=X+8:IF X>24 THEN X
    =0
960 IF A=11 THEN X=X-8:IF X<0 THEN X
    =24
970 IF A=14 THEN Y=Y-4:IF Y<20 THEN
    Y=48
980 IF A=13 THEN Y=Y+4:IF Y>48 THEN
    Y=20
990 POKE 53251,X+56:A$=O$(81-Y,81-Y+
    128)
1000 IF B=1 AND A<>15 THEN GOSUB 50
1010 IF PEEK(764)<>255 THEN GOSUB 10
     50
1020 IF PEEK(53279)=5 THEN FOR D=0 T
     O 3:POKE DL+D+19,4*(PEEK(DL+D+1
     9)=2)+2*(PEEK(DL+D+19)=4):NEXT
     D
1030 IF B=0 THEN GOSUB 210
1040 GOTO 930
1050 REM  KEYBOARD CHECK 
1060 GOSUB 140:GET #3,KEY:IF KEY>127
      THEN KEY=KEY-128:POKE 694,0
1070 A=KEY:A=A+64*(A<32)-32*(A>95)
1080 IF A=76 THEN GOSUB 780:GOSUB 60
1090 IF A=83 THEN GOSUB 850:GOSUB 60
1100 IF A=69 THEN GOSUB 290
1110 IF A=82 THEN GOSUB 330
1115 IF A=67 THEN GOSUB 1630
1120 IF A=84 THEN GOSUB 580:GOSUB 60
1130 IF A=77 OR A=32 THEN GOSUB 60
1140 IF KEY=49 THEN C=1
1150 IF KEY=50 THEN C=2
1160 IF KEY=51 THEN C=3
1170 POKE 764,255:RETURN
1180 REM  INITILIZATION 
```

```
1190 RAMTOP=PEEK(106)-12:POKE 89,RAM
     TOP:POKE 88,0:? CHR$(125):C=1:C
     HR=65:OPEN #3,4,0,"K:"
1200 POKE 106,RAMTOP:CHBASE=(RAMTOP+
     8)*256:PMBASE=(RAMTOP+4)*256:GR
     APHICS 0:POKE 710,176
1210 POKE 203,CHBASE/256:POSITION 9,
     3:? " INITIALIZING PROGRAM";:GO
     SUB 350
1220 FOR A=0 TO 24:READ B:POKE 1536+
     A,B:NEXT A:POKE 512,0:POKE 513,
     6
1230 VT=PEEK(134)+256*PEEK(135):AT=P
     EEK(140)+256*PEEK(141)
1240 X=CHBASE-AT:Y=57344-AT:GOSUB 13
     20.00005>L1149 POKE VT+2,X2:POK
     E VT+3,X1:POKE VT+4,1:POKE VT+5
     ,4:POKE VT+6,1:POKE VT+7,4
1260 POKE VT+10,Y2:POKE VT+11,Y1:POK
     E VT+12,1:POKE VT+13,4:POKE VT+
     14,1:POKE VT+15,4:A$=O$
1270 X=PMBASE+896-AT:Y=PMBASE-AT:GOS
     UB 1320:POKE VT+2,X2:POKE VT+3,
     X1:POKE VT+10,Y2:POKE VT+11,Y1
1280 X=PEEK(88)+256*PEEK(89)+100-AT:
     Y=RAMTOP*256-AT:GOSUB 1320:POKE
      VT+18,X2:POKE VT+19,X1:POKE VT
     +26,Y2
1290 POKE VT+27,Y1:X=PEEK(88)+256*PE
     EK(89)-AT:GOSUB 1320:POKE VT+34
     ,X2:POKE VT+35,X1
1300 X=PEEK(88)+256*PEEK(89)+380-AT:
     GOSUB 1320:POKE VT+42,X2:POKE V
     T+43,X1
1310 FOR A=4 TO 44 STEP 8:FOR B=0 TO
      3:READ D:POKE VT+A+B,D:NEXT B:
     NEXT A:GOTO 1330
1320 X1=INT(X/256):X2=INT(X-(256*X1)
     ):Y1=INT(Y/256):Y2=INT(Y-(256*Y
     1)):RETURN
1330 REM  PLAYER/MISSILE GRAPHICS
1340 POKE 54279,PMBASE/256:FOR D=0 T
     O 2:POKE 704+D,6:NEXT D:POKE 70
     7,68
1350 FOR D=53248 TO 53255:READ X:POK
     E D,X:NEXT D:FOR D=53256 TO 532
     58:POKE D,1:NEXT D:POKE 53259,3
     :X=0
```

```
1360 FOR A=0 TO 256 STEP 128:FOR D=2
     0 TO 52 STEP 4:POKE PMBASE+512+
     A+D,21:NEXT D:NEXT A
1370 FOR D=22 TO 50 STEP 4:POKE PMBA
     SE+384+D,85:NEXT D:POKE 623,17
1380 Y=20:FOR D=0 TO 3:POKE PMBASE+D
     +Y+896,3:NEXT D:FOR D=0 TO 3:PO
     KE PMBASE+D+80,3:NEXT D
1390 REM  ESTABLISH DISPLAY SCREEN 
1400 GRAPHICS 0:POKE 752,1:POKE 711,
     68:DL=PEEK(560)+256*PEEK(561):H
     OME=PEEK(DL+4)+256*PEEK(DL+5):P
     OKE DL+3,72
1410 FOR D=0 TO 8:POKE DL+D+6,8:NEXT
      D:POKE DL+18,144:GOSUB 760:POK
     E 54286,192
1420 GOSUB 150:POSITION 4,0:POKE 766
     ,1:FOR F=0 TO 3:FOR D=0 TO 31:?
      CHR$(D+32*F);:NEXT D:? :NEXT F
     :POKE 766,0
1430 POKE 559,46:POKE 53277,3:GOSUB
     390:GOSUB 140:GOSUB 60:GOTO 930
1440 DATA 72,138,72,152,72,165,203,1
     41,10,212,141,9,212,104,168,104
     ,170,104,64,104,162,16,76,86,22
     8
1450 DATA 128,0,128,0,128,1,128,1,12
     0,0,120,0,160,0,160,0,100,0,100
     ,0,160,0,160,0
1460 DATA 149,161,173,56,83,91,99,10
     7
1500 B=STRIG(1)*4:D=PEEK(53279)=3:IF
      A=7 THEN A=12
1510 A=A-10:IF A<0 THEN RETURN
1520 A=A+B:ON A GOTO 1530,1540,1550,
     1560,1570,1580,1590,1610:RETURN

1530 POKE 712,PEEK(712)-2+256*(PEEK(
     712)<2):RETURN
1540 POKE 712,PEEK(712)+2-256*(PEEK(
     712)>252):RETURN
1550 POKE 710,PEEK(710)-2+256*(PEEK(
     710)<2):RETURN
1560 POKE 710,PEEK(710)+2-256*(PEEK(
     710)>252):RETURN
1570 POKE 709,PEEK(709)-2+256*(PEEK(
     709)<2):RETURN
1580 POKE 709,PEEK(709)+2-256*(PEEK(
     709)>252):RETURN
```

```
1590 IF  NOT D THEN POKE 708,PEEK(70
     8)-2+256*(PEEK(708)<2):RETURN
1600 POKE 711,PEEK(711)-2+256*(PEEK(
     710)<2):RETURN
1610 IF  NOT D THEN POKE 708,PEEK(70
     8)+2-256*(PEEK(708)>252):RETURN

1620 POKE 711,PEEK(711)+2-256*(PEEK(
     711)>252):RETURN
1630 GOSUB 110:GOSUB 340:POSITION 8,
     1:? "Select character to copy"
1640 KEY=CHR:GET #3,CHR:GOSUB 390
1650 FOR A=0 TO 7:POKE CHBASE+KEY*8+
     A,PEEK(CHBASE+CHR*8+A):NEXT A:G
     OSUB 140:GOTO 70
```

4

Animation

4

Animation by Page Flipping

David N. Plotkin

A special animation technique, storing several predrawn screens in memory at once, summons them to create movement, flashing "frames" on the screen as movie projectors do. This article includes a simple game, "Inferno," to show how it's done.

Have you ever wished you could make a picture simply appear on the screen, without drawing it line by line in front of the user? Or perhaps you have an animation sequence, and drawing each new "frame"— erasing the parts you don't need, and PLOTting and DRAWTOing the new parts— takes too long and ruins the animated nature of the program. Well, the Atari computers do provide a way to draw a picture in the memory while the user is looking at a different picture, and then instantly flash the completed picture on the screen. By drawing several pictures beforehand in memory, and then flashing them on the screen one by one in sequence, you can easily create animation for your BASIC program.

The key to page flipping is that you can write to an area of memory that you are not displaying on the screen. When you then tell the computer to display the area of memory you have previously written to, the picture drawn in that area of memory simply appears on the screen.

Whenever you issue a GRAPHICS command, the computer creates what is called a *display list*. This is just information telling the computer how to display data on the screen. The memory address of the display list is stored at locations 560 and 561 as:

DL Address = PEEK (560) + 256* PEEK (561)

The fifth and sixth numbers of the display list (DL address + 4 and DL address + 5) contain the address of *screen memory*, that is, the address of the first byte of data to be displayed on the screen.

An entirely different set of memory locations contains the address of *write memory*, the memory address where the first byte of data is to be written from execution of keyboard commands and/or a running program:

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

Thus, when you type in a PLOT or DRAWTO command, the shape is written into computer memory at the location starting at the write memory address. The reason you normally see on the screen what has been written into write memory is that the address of write memory and the address of screen memory (or display memory) normally have the same value. To flip pages, then, you follow these steps:

1. POKE a value into locations 88 and 89 (write memory) which correspond to an empty, protected area of memory (more on this in a moment).

2. Execute PRINT, PLOT, DRAWTO, SETCOLOR, etc., commands, as usual, to draw the picture you want, using standard cursor limits. You will not see your picture on the screen.

3. Set DL address +4 and DL address +5 equal to the value in locations 88 and 89, respectively. Your picture will flash onto the screen.

Animating is just an extension of this method. You follow steps 1 and 2 listed above over and over, each time creating in memory a new "frame" of the animation. You have to keep track of where each new frame begins (the value of memory locations 88 and 89 for each screen) and make sure the screens don't overlap in memory. To prevent screens from overlapping, make sure that each screen starts further away in memory from the previous screen than the values listed below:

GRAPHICS 0 - 960 bytes	**GRAPHICS 5 - 960 bytes**
GRAPHICS 1 - 480 bytes	**GRAPHICS 6 - 1920 bytes**
GRAPHICS 2 - 240 bytes	**GRAPHICS 7 - 3840 bytes**
GRAPHICS 3 - 240 bytes	**GRAPHICS 8 - 7680 bytes**
GRAPHICS 4 - 480 bytes	

These values are just the amount of memory used to store one full screen of data. Remember that a change of the value of either location 89 or DL address + 5 by one is equal to 256 bytes, and that locations 88 and DL address + 4 cannot exceed 255. You'll need to do some math so that whenever location 88 exceeds 255, you subtract 256 from it and add one to location 89. Similarly, if location 88 goes below zero, then you add 256 to it and subtract one from location 89. These same rules apply for DL address + 4 or DL address + 5.

A couple of other pointers before we get to the fun part. Player/missile graphics are the best way to handle user-manipulated shapes when animating using page flipping. P/M graphics are not affected by the display memory or write memory manipulations. Assuming you left the write memory address pointing to one of your frames (usually the last one you drew), any further BASIC graphics or text commands during the program run will flash on the screen only when the single frame to which the command was written is put on the screen. To avoid this, you have to keep changing the write memory to match the screen memory, which you are changing to "flip pages," and create the animation effect. This takes time and slows down program execution. To produce the P/M graphics for the included program, I've used Eric Stoltman's machine language utility from the article "Extending Player/Missile Graphics" *(COMPUTE!'s First Book of Atari Graphics)* with a new wrinkle.

To move vertically (see lines 150-170), I first read zeros into the P/M memory, increment the Y coordinate, and then read the correct shape back into memory. This occurs quickly and works well. One last thing—to get the empty, protected memory to store your pictures, step back RAMTOP as many pages as you need and read zeros into the protected memory to clear it. See program lines 900-910 for details.

"Inferno" demonstrates page-flipping animation. A large skyscraper is burning fiercely, and the only escape for the 20 occupants is the roof. It's up to you to pilot your helicopter to a safe landing on the roof, navigating through the flames, which move faster as the fire progresses. The flames aren't as bad on the left side of the building, but that is where the fire engine is coming, so you can't land there. A flashing red dot appears in the upper left of the screen for each person you successfully rescue. When you've lost three helicopters or rescued all 20 people, just press the fire button to play again.

Variables

DIF:	Measures difficulty. As DIF increases, flames move faster
F:	Helicopter facing flag; = 1 when copter faces right; = −1 when copter faces left
P:	Person flag; = 1 when person on roof; = 0 when person rescued
NUM:	Number of people rescued
H:	Number of helicopters left
DL4:	Low byte of screen memory
DL5:	High byte of screen memory
ST:	STICK (0)
PB:	Start of P/M graphics memory
X0,Y0:	Player 0 coordinates
X1,Y1:	Player 1 coordinates
DL:	Address of display list
RT:	RAMTOP
A, I, N, M:	Loop variables

Program Listing

LINE	
10	Call initializing subroutines, initialize variables
20 – 80	Main Program Loop—set display memory address and jump to rescue subroutine
110 – 145	Move helicopter horizontally, keep it from going offscreen, set direction of helicopter
150 – 170	Move helicopter vertically
180	Test for helicopter landing on roof
190	Test for collision of helicopter with playing field
210	Test for dropping off passenger
230 – 250	Person runs to helicopter after it lands on roof
260 – 320	Destruction of helicopter, test for all helicopters destroyed
330 – 370	Passenger leaves helicopter after it lands. If all people are rescued, restart game. Increase difficulty as fire progresses
800 – 890	POKE machine language routine
900 – 980	Set up pages
1000 – 1370	Flip pages
1500 – 1550	Introduction

Inferno

```
10 GOSUB 15ØØ:DIF=Ø:F=-1:P=1:VEL=2:N
   UM=Ø:H=3:GOSUB 9ØØ:GOSUB 8ØØ:SOUN
   D 1,1ØØ,4,2
2Ø POKE DL5,RT:GOSUB 1ØØ
3Ø POKE DL5,RT+4:GOSUB 1ØØ
4Ø POKE DL5,RT+8:GOSUB 1ØØ
5Ø POKE DL5,RT+12:GOSUB 1ØØ
6Ø POKE DL5,RT+8:GOSUB 1ØØ
7Ø POKE DL5,RT+4:GOSUB 1ØØ
8Ø GOTO 2Ø
1ØØ FOR N=1 TO 5-DIF:ST=STICK(Ø):IF
    ST=16 THEN FOR WAIT=1 TO 15:NEXT
     WAIT:GOTO 185
11Ø XØ=XØ+VEL*(ST=7)*(XØ<2Ø1)-VEL*(S
    T=11)*(XØ>47)
12Ø IF XØ<48 THEN XØ=48
13Ø IF ST=7 AND F=-1 THEN F=1:D=USR(
    1536,PB+512+YØ,268):FOR W=1 TO 2
    Ø:NEXT W:D=USR(1536,PB+512+YØ,26
    Ø):XØ=XØ-2
14Ø IF ST=11 AND F=1 THEN F=-1:D=USR
    (1536,PB+512+YØ,268):FOR W=1 TO
    2Ø:NEXT W:D=USR(1536,PB+512+YØ,2
    76):XØ=XØ+2
145 POKE 53248,XØ
15Ø IF ST=14 THEN D=USR(1536,PB+512+
    YØ,292):YØ=YØ-2*(YØ>2Ø)
16Ø IF ST=13 THEN D=USR(1536,PB+512+
    YØ,292):YØ=YØ+2*(YØ<1Ø2)
17Ø D=USR(1536,PB+512+YØ,26Ø*(F=1)+2
    76*(F=-1))
18Ø IF XØ>112 AND XØ<138 AND YØ=28 A
    ND P=1 THEN GOSUB 23Ø
185 NEXT N
19Ø IF PEEK(53252)<>Ø THEN GOSUB 26Ø
21Ø IF YØ=1Ø2 AND P=Ø THEN GOSUB 33Ø
22Ø RETURN
23Ø N=(XØ>X1)-(X1>XØ):IF N=Ø THEN GO
    TO 25Ø
24Ø FOR M=X1 TO XØ STEP N:POKE 53249
    ,M:NEXT M:X1=M
25Ø D=USR(1536,PB+64Ø+Y1,292):P=Ø:RE
    TURN
26Ø POKE 53278,1:D=USR(1536,PB+512+Y
    Ø,3ØØ):FOR W=12Ø TO 4Ø STEP -1:P
    OKE 7Ø4,W
27Ø POKE PB+512+YØ+INT(RND(Ø)*7),PEE
    K(5377Ø):SOUND Ø,2ØØ,8,W/1Ø:NEXT
     W:SOUND Ø,Ø,Ø,Ø:POKE 7Ø4,25Ø
```

```
280 D=USR(1536,PB+512+YØ,292):H=H-1:
    POSITION 29,Ø:? H;:IF H<>Ø THEN
    GOTO 3ØØ
290 POKE DL5,RT+12:POSITION Ø,Ø:? "G
    AME OVER-RESCUED ";NUM;:POSITION
     2Ø,Ø:? "  PRESS FIRE ";
291 IF STRIG(Ø)=1 THEN 291
292 POSITION Ø,Ø:? "{37 SPACES}";
295 H=3:COLOR Ø:NUM=Ø:DIF=Ø:VEL=2:GO
    SUB 97Ø
300 POKE 53278,1:XØ=18Ø:YØ=2Ø:POKE 5
    3248,XØ:D=USR(1536,PB+512+YØ,276
    ):F=-1
310 IF P=Ø THEN X1=12Ø:Y1=28:POKE 53
    249,X1:D=USR(1536,PB+64Ø+Y1,284)
    :P=1
320 RETURN
330 D=USR(1536,PB+64Ø+YØ,284)
340 FOR M=XØ TO 21Ø:POKE 53249,M:NEX
    T M:NUM=NUM+1:D=USR(1536,PB+64Ø+
    YØ,292):IF NUM=2Ø THEN GOTO 37Ø
345 POSITION 9,Ø:? NUM;:IF NUM/4=INT
    (NUM/4) THEN DIF=DIF+1:IF NUM/8=
    INT(NUM/8) THEN VEL=VEL+2
350 P=1:POKE 53249,X1:D=USR(1536,PB+
    64Ø+Y1,284)
360 RETURN
370 D=USR(1536,PB+512+YØ,292):GOTO 29Ø
800 FOR A=1536 TO 156Ø:READ I:POKE A
    ,I:NEXT A
810 DATA 1Ø4,1Ø4,133,2Ø4,1Ø4,133,2Ø3
    ,1Ø4,133,2Ø7,1Ø4,133,2Ø6,16Ø,Ø,1
    77,2Ø6,145,2Ø3,2ØØ,192,8,2Ø8,247,96
820 FOR A=26Ø TO 3Ø7:READ I:POKE A,I
    :NEXT A
830 DATA Ø,31,4,143,249,15,2,63,Ø,12
    7,8,62,34,62,2Ø,62,Ø,248,32,241,
    159,24Ø,32,252
840 DATA Ø,Ø,24,24,126,24,36,1Ø2,Ø,Ø
    ,Ø,Ø,Ø,Ø,Ø,Ø,68,186,84,13Ø,255,1
    3Ø,68,186
850 POKE 752,1:POKE 559,46
860 A=PEEK(1Ø6)-4:POKE 54279,A:POKE
    53277,3:PB=256*A:XØ=18Ø:YØ=2Ø:X1
    =12Ø:Y1=28
875 POKE 7Ø4,25Ø:POKE 7Ø5,8Ø
880 POKE 53248,XØ:D=USR(1536,PB+512+
    YØ,276):POKE 53249,X1:D=USR(1536
    ,PB+64Ø+Y1,284)
890 RETURN
```

```
900 RT=PEEK(1Ø6)-16:GRAPHICS 5+16:PO
    KE 88,Ø:POKE 89,RT-4:? #6;CHR$(1
    25):POKE 1Ø6,RT:GRAPHICS 5+16
910 POKE 559,Ø
915 SETCOLOR 4,7,4:SETCOLOR 1,12,8:S
    ETCOLOR 2,3,3
920 PAGE=Ø:POKE 88,Ø:POKE 89,RT:GOSU
    B 1ØØØ
930 PAGE=1:POKE 88,Ø:POKE 89,RT+4:GO
    SUB 1ØØØ
940 PAGE=2:POKE 88,Ø:POKE 89,RT+8:GO
    SUB 1ØØØ
950 PAGE=3:POKE 88,Ø:POKE 89,RT+12:G
    OSUB 1ØØØ
960 DL=PEEK(56Ø)+256*PEEK(561):DL4=D
    L+4:DL5=DL+5:POKE DL4,Ø:POKE DL5
    ,RT+4:POKE 559,34
970 POKE 752,1:POKE DL+3,66:POKE 87,
    Ø:POSITION Ø,Ø:? "RESCUED :";NUM
    ;:POSITION 2Ø,Ø:? "COPTERS :";H;
980 RETURN
1000 COLOR 1:FOR X=32 TO 47:PLOT X,1
     Ø:DRAWTO X,47:NEXT X
1010 COLOR 2:FOR X=33 TO 45 STEP 4:F
     OR Y=13 TO 41 STEP 4:PLOT X,Y:P
     LOT X+1,Y:PLOT X,Y+1:PLOT X+1,Y
     +1:NEXT Y:NEXT X
1020 PLOT 39,45:DRAWTO 39,47:PLOT 4Ø
     ,45:DRAWTO 4Ø,47:PLOT Ø,47:DRAW
     TO 31,47:PLOT 48,47:DRAWTO 79,4
     7
1030 IF PAGE=1 OR PAGE=2 OR PAGE=3 T
     HEN GOTO 11ØØ
1040 COLOR 3:FOR Y=45 TO 46:PLOT 1,Y
     :DRAWTO 8,Y:NEXT Y:PLOT 2,47:PL
     OT 7,47:COLOR Ø:PLOT 5,45
1050 RETURN
1100 COLOR 3:PLOT 48,13:PLOT 48,14:P
     LOT 49,12:PLOT 48,26:PLOT 48,27
     :PLOT 49,25:PLOT 48,36:PLOT 48,
     37:PLOT 49,35
1110 PLOT 31,19:DRAWTO 31,21:PLOT 3Ø
     ,18:PLOT 31,31:DRAWTO 31,33:PLO
     T 3Ø,3Ø
1120 IF PAGE=2 OR PAGE=3 THEN GOTO 1
     2ØØ
1130 FOR Y=45 TO 46:PLOT 14,Y:DRAWTO
      21,Y:NEXT Y:PLOT 15,47:PLOT 2Ø
     ,47:COLOR Ø:PLOT 18,45
1140 RETURN
```

```
1200 COLOR 3:PLOT 48,11:PLOT 48,12:P
     LOT 49,11:PLOT 49,10:PLOT 50,10
     :PLOT 50,9
1210 PLOT 48,24:PLOT 48,25:PLOT 49,2
     4:PLOT 49,23:PLOT 50,23:PLOT 50
     ,22
1220 PLOT 48,35:PLOT 48,34:PLOT 49,3
     4:PLOT 49,33:PLOT 50,33:PLOT 50
     ,32
1230 PLOT 31,40:PLOT 31,41:PLOT 31,3
     0:PLOT 31,29:PLOT 30,29:DRAWTO
     30,27:PLOT 29,27:PLOT 29,26
1240 PLOT 31,18:PLOT 31,17:PLOT 30,3
     9:PLOT 30,17:DRAWTO 30,15:PLOT
     29,15:PLOT 29,14
1250 IF PAGE=3 THEN GOTO 1300
1260 FOR Y=45 TO 46:PLOT 24,Y:DRAWTO
      31,Y:NEXT Y:PLOT 25,47:PLOT 30
     ,47:COLOR 0:PLOT 28,45
1270 RETURN
1300 COLOR 3:PLOT 49,9:DRAWTO 49,3:P
     LOT 50,8:DRAWTO 50,5:PLOT 51,7:
     DRAWTO 51,9
1310 PLOT 49,22:PLOT 49,21:PLOT 50,2
     1:DRAWTO 50,19:PLOT 51,20:DRAWT
     O 51,22
1320 PLOT 49,32:PLOT 49,31:PLOT 50,3
     1:DRAWTO 50,29:PLOT 51,30:DRAWT
     O 51,32
1330 PLOT 30,38:PLOT 30,37:PLOT 29,3
     8:DRAWTO 29,35
1340 PLOT 30,26:DRAWTO 30,24:PLOT 29
     ,25:DRAWTO 29,22
1350 PLOT 30,14:DRAWTO 30,12:PLOT 29
     ,13:DRAWTO 29,10
1360 COLOR 3:FOR Y=45 TO 46:PLOT 1,Y
     :DRAWTO 8,Y:NEXT Y:PLOT 2,47:PL
     OT 7,47:COLOR 0:PLOT 5,45
1370 RETURN
1500 GRAPHICS 2+16:POSITION 0,1:? #6
     ;"COMPUTE PUBLICATIONS":POSITIO
     N 6,3:? #6;"PRESENTS"
1510 POSITION 6,6:? #6;"inferno"
1530 FOR SND=20 TO 120 STEP 0.2:SOUN
     D 0,SND,8,6:POKE 712,SND:NEXT S
     ND
1540 FOR VOL=6 TO 0 STEP -0.1:SOUND
     0,SND,8,VOL:NEXT VOL:POKE 712,0
1550 RETURN
```

4

Player/Missile Graphics Simplified

Staffan Sandberg

You've seen the wonderful things the Atari can do with player/missile graphics, but until now you've either had to settle for slow-moving wobbles or learn machine language. Here is an overlay method which is simple to use and results in extremely fast animation of up to five players.

In the overlay method we will design overlays, or patterns that we can place on the screen. We can create as many patterns as we want and use them as often as we want. Each overlay is eight dots wide and anything from one to 128 dots high. The overlay allows specified dots to be lit up on the screen. When we want an object to appear to be moving, we place one of the overlays on the screen by specifying its X and Y coordinates. We then give it new X and Y coordinates, and it appears to move. This process is very fast, so the object appears to move quite quickly. These overlays are totally separate from player/missile graphics. It is the combination of the overlays and player/missile graphics that allows us the freedom of movement of the overlay method.

To use overlays, just follow these steps:

Step 1: Decide how many players you wish to use and set aside enough memory to hold them. That is, what is the maximum number of objects you want on the screen at one time? You can have up to five. We must give each one a name and set aside 128 spaces for it because each player is potentially 128 dots high. We do this by DIMensioning the space:

```
10 DIM PM1$(128),PM2$(128),PM3$(128)
```

The DIMensioning must be the first thing the computer sees when it is turned on, so before you start programming, turn off the computer and turn it back on. This is necessary because as the computer constructs a variable table, the variables are stored in the order that they are entered. The variable table is not cleared by typing NEW. We want these variables at the beginning of the table so we can find them easily later. If they are not the first thing that the computer sees, the method will not work.

Step 2: Design the overlays or patterns that you wish to use. Remember, you can create as many overlays as you wish. They are stored in strings (ALIEN$, SHIP$, etc.), so you must give each overlay a name and DIMension its size. When deciding the size of each overlay, keep the following questions in mind:

1. How tall do you want to make your overlay?
2. What directions do you want to move your player?
3. How fast do you want to move your players?

You don't need to worry about the width of the overlay. But you must decide how many dots high you wish to make an overlay. It can be up to 128 dots in height (an average spaceship might be six dots high). If you are going to be moving your players down the screen , you must leave blank spaces to cover up the old overlay, and you must take into account the speed at which your player will move. The speed is measured in Dots Per Move (DPM). If your players will be moving at a top speed of three DPM up and down the screen, then you need to leave three spaces above and three spaces below. To help decide the size to be DIMensioned for each overlay, use the formula:

SIZE = height of overlay + DPM up + DPM down

SHIP$ and ALIEN$ Examples

In our example we will have one ship which we'll call SHIP$, with a height of six moving up and down at the speed of five DPM, and another ship which we'll call ALIEN$, with a height of eight moving neither up nor down.

```
20 SIZE1=16:SIZE2=8
30 DIM SHIP$(SIZE1),ALIEN$(SIZE2)
```

We also want a blank overlay that we use to erase the player from the screen quickly. We'll call this overlay CLEAR$. It should be 128 dots high so that it can erase anything on the 128 dot high player.

```
40 DIM CLEAR$(128)
```

Now you must create the overlays line by line. Each line or row is made up of dots or "boxes." Each box is numbered from right to left 1, 2, 4, 8, 16, 32, 64, and 128.

128	64	32	16	8	4	2	1

To create the overlays, you must decide which boxes you want filled or lit up on the screen. You then add the value of each filled box for each row (see Figures 1 and 2).

Figure 1. The Ship

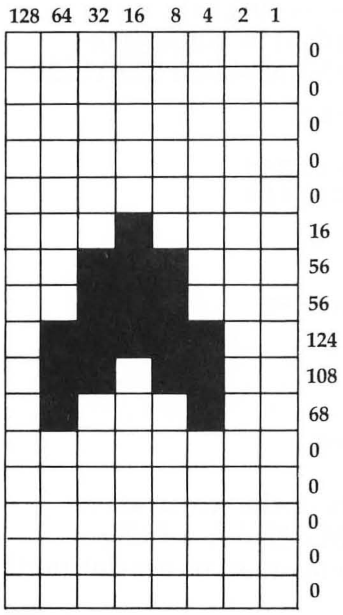
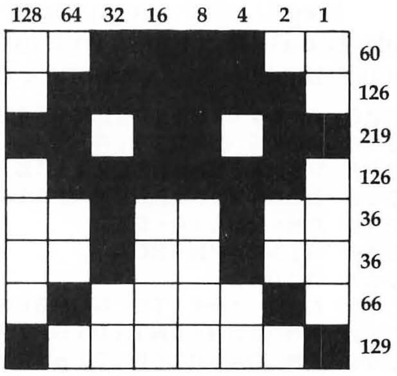


Figure 2. The Alien



Now that you have the totals for each row, you must put them in the string that you have DIMensioned for them. This is done in a short loop such as the one below.

```
50 FOR ROWS=1 TO SIZE1
60 READ DOTS
70 SHIP$(ROWS,ROWS)=CHR$(DOTS)
80 NEXT ROWS
90 DATA 0,0,0,0,0
100 DATA 16,56,56,124,108,68
110 DATA 0,0,0,0,0
120 FOR ROWS=1 TO SIZE2
130 READ DOTS
```

```
140 ALIEN$(ROWS,ROWS)=CHR$(DOTS)
150 NEXT ROWS
160 DATA 60,126,219,126,36,36,66,129
```

You need a loop for each overlay that you have.

You also need to create the blank overlay, CLEAR$, by entering 128 blank lines into CLEAR$.

```
170 FOR ROWS=1 TO 128
180 CLEAR$(ROWS,ROWS)=CHR$(0)
190 NEXT ROWS
```

Step 3: Tell the computer that you are going to be using player/missile graphics with overlay method by entering the following lines, substituting a value for NUMBEROFPLAYERS.

```
200 A=4*(INT(PEEK(742)/4)-1)
210 POKE 54279,A
220 VSA=256*PEEK(135)+PEEK(134)
230 BOA=256*PEEK(141)+PEEK(140)
240 PM=256*A+512
250 DISP=PM-BOA
260 ADD=2
270 FOR T=1 TO NUMBEROFPLAYERS
280 PMHIGH=INT(DISP/256)
290 PMLOW=DISP-256*PMHIGH
300 POKE VSA+ADD,PMLOW
310 POKE VSA+ADD+1,PMHIGH
320 DISP=DISP+128:ADD=ADD+8
330 NEXT T
```

If you are going to have five players on the screen at one time, you must change line 240 from PM = 256* A + 512 to PM = 256* A + 384. This tells the computer to let us use the fourth missile as a player.

Step 4: Now we are ready to add the initial specifications, such as color, size, and shape to the players. First, line 340 places the blank overlay on each player, clearing out any stray data.

```
340 PM1$=CLEAR$:PM2$=CLEAR$:PM3$=CLEAR$
```

Next we set the player/missile graphics to double-line resolution and turn on the P/M graphics (a 3 enables them and a 0 disables them).

```
350 POKE 559,46:POKE 53277,3
```

To set the colors of the players, we must POKE the color register for each player with the proper color number. The registers go from 704 (for Player 0) through 707 (for Player 3). The fifth player takes on a combination of the colors of the other four. The colors that I have chosen are: COLR1 is yellow, COLR2 is white, and COLR3 is pink.

```
360 COLR1=25:COLR2=11:COLR3=74
370 POKE 704,COLR1:POKE 705,COLR2:PO
    KE 706,COLR3
```

The size of the players is automatically set to normal. If you want to change the size, POKE 0 for normal, 1 for double, and 3 for quadruple size into the size register for the corresponding player. These registers go from 53256 (for Player 0) through 53259 (for Player 3).

POKE 53256, 1 would make Player 0 double size.

Now we can place the player on the screen. First, we give the player an X (horizontal) value and POKE it into the horizontal position register for each player. The registers go from 53248 (for Player 0) through 53251 (for Player 3). The horizontal positions that show up on the screen range from about 50 to 200 (depending on your TV). Numbers lower than 50 and greater than 200 are to the right and left of the screen.

```
380 X1=125:X2=75:X3=175
390 POKE 53248,X1:POKE 53249,X2:POKE
     53250,X3
```

Now we must give our player a Y (vertical) value and an overlay. The format is PM$ (Y value) = overlay.

```
400 Y1=125:Y2=25:Y3=25
410 PM1$(Y1)=SHIP$:PM2$(Y2)=ALIEN$:P
    M3$(Y3)=ALIEN$
```

To move the player around the screen, change the X and/or the Y value and repeat steps 390 and 410. Be sure not to change the X value more than the maximum DPM that you decided earlier. If you do, you will leave parts of the overlay on the screen.

Program 1. Player/Missile Graphics Example 1

```
5 REM PMSAMPLE1
10 DIM PM1$(128),PM2$(128),PM3$(128)
20 SIZE1=16:SIZE2=8
30 DIM SHIP$(SIZE1),ALIEN$(SIZE2)
40 DIM CLEAR$(128)
50 FOR ROWS=1 TO SIZE1
60 READ DOTS
70 SHIP$(ROWS,ROWS)=CHR$(DOTS)
```

```
80 NEXT ROWS
90 DATA Ø,Ø,Ø,Ø,Ø
100 DATA 16,56,56,124,108,68
110 DATA Ø,Ø,Ø,Ø,Ø
120 FOR ROWS=1 TO SIZE2
130 READ DOTS
140 ALIEN$(ROWS,ROWS)=CHR$(DOTS)
150 NEXT ROWS
160 DATA 6Ø,126,219,126,36,36,66,129
170 FOR ROWS=1 TO 128
180 CLEAR$(ROWS,ROWS)=CHR$(Ø)
190 NEXT ROWS
200 A=4*(INT(PEEK(742)/4)-1)
210 POKE 54279,A
220 VSA=256*PEEK(135)+PEEK(134)
230 BOA=256*PEEK(141)+PEEK(14Ø)
240 PM=256*A+512
250 DISP=PM-BOA
260 ADD=2
270 FOR T=1 TO 3
280 PMHIGH=INT(DISP/256)
290 PMLOW=DISP-256*PMHIGH
300 POKE VSA+ADD,PMLOW
310 POKE VSA+ADD+1,PMHIGH
320 DISP=DISP+128:ADD=ADD+8
330 NEXT T
340 PM1$=CLEAR$:PM2$=CLEAR$:PM3$=CLE
    AR$
350 POKE 559,46:POKE 53277,3
360 COLR1=25:COLR2=11:COLR3=74
370 POKE 7Ø4,COLR1:POKE 7Ø5,COLR2:PO
    KE 7Ø6,COLR3
380 X1=125:X2=75:X3=175
390 POKE 53248,X1:POKE 53249,X2:POKE
     5325Ø,X3
400 Y1=75:Y2=25:Y3=25
410 PM1$(Y1)=SHIP$:PM2$(Y2)=ALIEN$:P
    M3$(Y3)=ALIEN$
420 IF STICK(Ø)<8 THEN X1=X1+3
430 IF STICK(Ø)>8 AND STICK(Ø)<13 TH
    EN X1=X1-3
440 IF STICK(Ø)=14 THEN Y1=Y1-3
450 IF STICK(Ø)=13 THEN Y1=Y1+3
460 POKE 53248,X1:PM1$(Y1)=SHIP$
470 IF STICK(1)<8 THEN X2=X2+2
480 IF STICK(1)>8 AND STICK(1)<13 TH
    EN X2=X2-2
490 POKE 53249,X2
500 IF STICK(2)<8 THEN X3=X3+2
```

```
510 IF STICK(2)>8 AND STICK(2)<13 TH
    EN X3=X3-2
520 POKE 53250,X3
530 GOTO 420
```

Program 2. Player/Missile Graphics Example 2

```
5 REM PMSAMPLE2
10 DIM PM$(128)
20 DIM SHIP$(16),CLEAR$(128)
30 FOR ROW=1 TO 16
40 READ DOTS
50 SHIP$(ROW,ROW)=CHR$(DOTS)
60 NEXT ROW
70 DATA 0,0,0,0,0
80 DATA 16,56,56,124,108,68
90 DATA 0,0,0,0,0
100 FOR ROW=1 TO 128
110 CLEAR$(ROW,ROW)=CHR$(0)
120 NEXT ROW
130 A=4*(INT(PEEK(742)/4)-1)
140 POKE 54279,A
150 VSA=256*PEEK(135)+PEEK(134)
160 BOA=256*PEEK(141)+PEEK(140)
170 PM=256*A+512
180 DISP=PM-BOA
190 ADD=2
200 FOR T=1 TO 1
210 PMHIGH=INT(DISP/256)
220 PMLOW=DISP-256*PMHIGH
230 POKE VSA+ADD,PMLOW
240 POKE VSA+ADD+1,PMHIGH
250 DISP=DISP+128:ADD=ADD+8
260 NEXT T
270 PM$=CLEAR$
280 POKE 559,46:POKE 53277,3
290 POKE 704,12
300 POKE 53248,50
310 PM$(10)=SHIP$
320 N=1
330 FOR X=60 TO 190 STEP N
340 POKE 53248,X
350 NEXT X
360 FOR Y=10 TO 100 STEP N
370 PM$(Y)=SHIP$
380 NEXT Y
390 FOR X=190 TO 60 STEP -N
400 POKE 53248,X
410 NEXT X
```

```
420 FOR Y=100 TO 10 STEP -N
430 PM$(Y)=SHIP$
440 NEXT Y
450 N=N+0.1
460 IF N>5 THEN N=1
470 GOTO 330
```

PRINTing P/M Graphics

Sheldon Leemon

Using a PRINT statement to make multiple POKEs can speed up your programs. The technique is easily learned and will work especially well with player/missile graphics.

The PRINT statement is one of the easiest BASIC commands to learn, and one of the first mastered by the beginning programmer. POKE, on the other hand, seems very mysterious to those just starting out with computers. Just changing the value of certain memory locations can have many different effects, but the how and why often elude the novice.

The POKE and PRINT commands are alike in a number of ways. For example, it is fairly easy to make a character appear on the screen by POKEing a value to display memory. Since Atari screen memory isn't always in the same place, you first have to find the start of screen memory by entering the statement SM = PEEK(88) + 256*PEEK(89). Now you POKE SM,33, and the letter A will appear in the top-left corner of the screen. You might think of this as PRINTing with the POKE statement. The reason that you use a 33 to make an A appear is that it is the internal screen code value used to display that letter. A table of these screen codes appears on page 55 of the *Atari BASIC Reference Manual.* (The internal character set can also be found on page 120 of *COMPUTE!'s First Book of Atari Graphics* and in *COMPUTE!'s Third Book of Atari.)* Try not to confuse this with the ATASCII values that appear in Appendix C of the manual. Those are the values that work with the CHR$ statement.

The fact that you can accomplish the same thing with both statements points out their fundamental similarity. When you get right down to it, all that the PRINT statement really does is to POKE a series of values into screen memory according to certain rules of cursor placement. The main advantage of using PRINT instead of POKE is its ease of use and speed, as compared to calculating a screen location and a numeric value for each character that you wish to display.

But if printing to the screen really involves only quickly POKEing a series of values into screen memory, it is not too different from a number of other situations in which the user must POKE a number of values into sequential memory locations. Examples of such situations include putting a machine language subroutine into place, installing a user-defined character set, or setting a number of music or graphics registers at once. If PRINT can facilitate the movement of a number of values to screen and color memory in one situation, might it not also be able to help out in some of these other situations?

Fooling the Computer

It is quite possible to use the PRINT statement for purposes other than printing to the screen. To do so, we have to make the computer think that the address we want POKEd to is the address of screen memory, and then PRINT a string of characters to that location whose POKE values correspond to the changes we wish to make to memory. The first part is possible because screen memory is not fixed in one set location in the Atari. Instead, the computer has a way of telling the display chip what spot in memory to display, and of letting the operating system know what area of memory is currently being shown on screen. Thus it knows the proper place to PRINT. Normally, the display chip reads and displays the same area of screen memory that the operating system (OS) writes. It is possible, however, to change the OS pointer so that the area of memory being PRINTed is different from the one being displayed.

Locations 88 and 89 hold the OS pointer to screen memory. The number in location 88 plus the number in location 89 multiplied by 256 equals the address of the first location which will be written to as screen memory. Let's try an experiment to see how this works. ENTER and RUN the following:

```
10 L=PEEK(88):H=PEEK(89)
20 POKE 88,0:POKE 89,6
30 POSITION 0,0:PRINT "A":PRINT "B"
40 POKE 88,L:POKE 89,H
```

Nothing is printed on the screen, although the cursor does move to the top line. Where in memory did the PRINT statements write to?

We can figure out step by step what locations must have changed. When we POKEd 88 with a 0 and 89 with a 6, we changed what the computer thought the top of PRINT memory was to 0 + 6*256 or 1536. Using the POSITION statement, we home the cursor to the top left, which is the first character in screen memory. Therefore, the screen value of A must have been POKEd into 1536. If we type:

```
PRINT PEEK(1536)   <RETURN>
```

We find a 33, which is indeed the screen value of A. But where did the B go? Since after every PRINT statement, the cursor moves to the next line, the cursor would move at least 40 spaces in memory. In addition, the cursor also moves two more spaces for the right margin, making a total of 42. If we type:

```
PRINT PEEK(1536+42) <RETURN>
```

We will find a 34, which is the screen value of B.

Using PRINT with P/M Graphics

One practical application for this technique is in displaying player/missile graphics. Normally, to move a player vertically you must POKE each byte of shape data into memory, one byte at a time, causing motion that is slow and jerky. But by changing the operating system PRINT pointers to the player area, we can use the speed of the PRINT statement to make the multiple POKEs appear to occur at the same time. This can be done by PRINTing a string of characters into player memory. This string will contain the data for the player. And, by combining data for several shapes into one long shape string, we can change the shape of the player at any time just by PRINTing a different segment of the string.

Program 1 shows just how easily a short BASIC program can move and animate a player. When the ship appears on the screen, insert a joystick into port 1. Use the joystick to move the ship around. You will notice that when you move the ship, its shape will change to point in the direction in which it is moving. The program has enough internal remarks to enable someone with a basic understanding of player/missile graphics to follow the program logic.

Player/Missile Graphics from PILOT

An added feature of this technique is that it is applicable to other languages besides BASIC. PILOT, for example, is a language that lacks built-in player/missile graphics commands. The T: statement in PILOT works the same way as PRINT in BASIC, so it is possible to use the same techniques to move a player in PILOT. Program 2 is a simplified program in PILOT which demonstrates moving a player with a joystick. Considering that PILOT allows only one statement per line, you can see that this program is extremely compact. When you type it in, pay careful attention to characters you enter for $SHIP. There are instructions in the comments at the end of the program to help you.

UnPRINTable Characters

One problem you should keep in mind is that not every value that you may need in screen memory can be placed there by a printing character. Some characters move the cursor rather than print anything on the screen, and in order to get them to print you must precede them with an ESCape character. For example, let's suppose you want to put into screen memory a value of 28, which corresponds to the cursor up arrow. Rather than moving the "cursor" up a line, you must print an ESCape and an up arrow in order for the character to be placed into memory. Program 1 has many instances in which such nonprinting characters are preceded by a 27 so that their number value will be placed in memory. Some characters, like the quote (internal value of 2) and the carriage return (internal value of 253) are even more difficult to PRINT. Keep a sharp eye out for such exceptions. Since perhaps the greatest drawback to this technique is in generating the strings to print, you might want to write a short program to do this for you.

Other Uses

I think you'll find this technique a useful one for controlling player/missile graphics from BASIC. But don't ignore the possibilities for using it to install nonrelocatable machine language into page six, for producing 16-bit sound, and for other applications where you have to move a number of bytes of data to a specific spot in memory quickly and efficiently.

Program 1. PRINTing Player/Missile Graphics from BASIC

```
1 GOTO 2Ø
2 POKE 88,Y:POSITION Ø,Ø:? SHIP$(I,I+15);
3 POKE 53248,X
4 GOTO STICK(Ø)
5 Y=Y+1:X=X+1:I=49:GOTO 2
```

```
6 Y=Y-1+(Y=Ø):X=X+1:I=17:GOTO 2
7 X=X+1:I=33:GOTO 2
9 Y=Y+1:X=X-1:I=81:GOTO 2
1Ø Y=Y-1+(Y=Ø):X=X-1:I=113:GOTO 2
11 X=X-1:I=97:GOTO 2
13 Y=Y+1:I=65:GOTO 2
14 Y=Y-1+(Y=Ø):I=1:GOTO 2
15 GOTO 4:REM Line 2 sets screen poi
   nter and prints ship into player
   Ø data area, line 3 moves ship ho
   rizontally
2Ø ? CHR$(125):POKE 752,1:POKE 82,Ø:
   POKE 83,39:? " ":REM clear screen
   , set margins to full screen, cur
   sor off
3Ø DIM BL$(255),SHIP$(128):BL$(1)="
   ":BL$(255)=" ":BL$(2)=BL$:REM fil
   l bl$ with space character
4Ø FOR I=1 TO 128:READ A:SHIP$(I,I)=
   CHR$(A):NEXT I:REM Read player sh
   ape data into ship$
5Ø DATA 32,56,56,56,56,56,92,92,92,2
   7,126,27,126,231,131,32
51 DATA 32,32,33,39,62,27,254,27,126
   ,46,44,44,6Ø,4Ø,4Ø,32
52 DATA 32,32,16Ø,128,224,112,124,95
   ,124,112,224,128,16Ø,32,32,32
53 DATA 32,32,4Ø,4Ø,6Ø,44,44,46,27,1
   27,27,254,62,39,33,32
54 DATA 32,131,231,27,126,27,126,92,
   92,92,56,56,56,56,56,32
55 DATA 32,32,48,48,88,8Ø,8Ø,112,27,
   126,27,127,12Ø,228,16Ø,32
56 DATA 32,32,33,35,39,46,94,252,94,
   46,39,35,33,32,32,32
57 DATA 32,32,16Ø,228,12Ø,27,127,27,
   126,112,8Ø,8Ø,88,48,48,32
6Ø A=PEEK(1Ø6)-16:POKE 54279,A:X=129
   :Y=96:POKE 7Ø4,14:REM set pmgraph
   ics base, x and y position and co
   lor of ship
7Ø POKE 88,Ø:POKE 89,A+4:POSITION Ø,
   Ø:? BL$:REM set screen pointer to
    player Ø data area, print blanks
    to clear
8Ø POKE 559,3Ø:POKE 53277,3:I=1:GOTO
    2:REM enable pmgraphics, go to m
   ove loop
```

Program 2. PRINTing Player/Missile Graphics from PILOT

```
100 J:*PMINIT
110 *MOVE C:@B53248=#X
120 C:@B88=#Y
130 POS:0,0
140 T:$SHIP
150 *LOOP A:$S=%J0
160 MS:10,9,8,6,5,4,2,1,0
170 JM:*DR,*UR,*R,*DL,*UL,*L,*D,*U,*
    LOOP
180 *DR C:#X=#X+1
190 *D C:#Y=#Y+1
200 J:*MOVE
210 *UR C:#Y=#Y-1
220 *R C:#X=#X+1
230 J:*MOVE
240 *DL C:#Y=#Y+1
250 *L C:#X=#X-1
260 J:*MOVE
270 *UL C:#X=#X-1
280 *U C:#Y=#Y-1
290 J:*MOVE
300 *PMINIT
310 T:{CLEAR}{13 SPACES}[CLEAR SCREEN
320 C:@82=9984{6 SPACES}[FULL MARGINS
330 C:$SHIP= 9]{ESC}{INSERT}]9 [PM S
    HAPE DATA
340 C:#P=@B106-12{3 SPACES}[BELOW SC
    REEN...
350 C:@B54279=#P{4 SPACES}[IS PM AREA
360 C:#X=128{8 SPACES}[PLAYER X POS.
370 C:#Y=48{9 SPACES}[PLAYER Y POS.
380 C:@B704=14{6 SPACES}[PLAYER IS W
    HITE
390 C:@B559=14{6 SPACES}[ENABLE DISP
    LAY..
400 C:@B53277=3{5 SPACES}[OF PM GRAP
    HICS
410 C:@B89=#P+2{5 SPACES}[SET PR. PO
    INTER
420 J:*MOVE
430 R:
440 R:BE CAREFUL IN TYPING LINE 330
450 R:AFTER THE EQUAL SIGN, TYPE:
460 R:SPACE,INVERSE KEY(ATARI LOGO)
470 R:NINE,RIGHT BRACKET,ESCAPE,ESCAPE
480 R:ESCAPE,CONTROL AND INSERT,
490 R:RIGHT BRACKET,9,INVERSE (LOGO),
500 R:SPACE,(COMMENT IS OPTIONAL)
```

5

Artists' Utilities

5

Fontbyter

Orson Scott Card and Carl Zahrt

"Fontbyter" is a utility which makes creating graphics displays in ANTIC modes 4 and 5 both easy and fun. Fontbyter requires a minimum of 40K memory.

It's hard to tell, when you're using "Fontbyter," whether this is a utility or a game. You can easily create graphics displays many times the size of the screen and save them to disk, using the ROM character set—or character sets you have designed yourself. And because Fontbyter allows you to use two "hidden" character modes, ANTIC modes 4 and 5, you get all the high-resolution color of GRAPHICS 7 with the convenience and memory usage of GRAPHICS 0.

Once you have a character set designed and a picture drawn on the screen, changing an 8-by-8-pixel character block takes only one POKE. This allows easy, almost instant animation; your programs can be shorter than they would be if you tried to get the same effect with GRAPHICS 7; and you have more memory available to you because the screen displays take up less room.

The problem is creating the actual display. In ANTIC 4, you have 24 lines of 40 characters; in ANTIC 5, 12 lines of 40 characters. Laying out the screen display and writing the DATA statements can be a long, tedious, painful process. You have to remember what each character looks like and make sure that the characters are in the right order in the DATA statements you create. And when you want to change a display, you have to go back and find the right DATA statement and alter it.

Fontbyter lets you create and edit in ANTIC 4 or 5 right on the screen. You don't have to write down the number of the character and POKE it into memory; you only have to press a key or combination of keys, and your character will be displayed exactly where you want it on the screen. Simple commands allow you to fill large areas with a single character, insert or delete lines, scroll around the screen to view large areas quickly, or change the colors on the screen. And Fontbyter will scroll horizontally and vertically, so you can use the screen as a window onto a very large display—up to 4K.

Best of all, you can save your screen to disk at any point and return to continue editing it. Using a simple subroutine, you can then load your screen into memory in your own program. The first eight bytes of every file Fontbyter creates contain the mode number, the display width, the display height, and the five colors of the screen display.

Starting Fontbyter

Character Set. When you RUN Fontbyter, the program accesses your disk and shows you a directory of all the files with the filetype ".SET". Fontbyter assumes that these are all character sets. The program then asks you to choose which one you want to use. Or, if you wish to use the built-in ROM character set, enter the character @ as the filename.

There is only one custom character set included with Fontbyter, but by using a character editor you can create as many different sets as you want. There are several character editor programs that will help you create your own character sets, including Tim Kilby's ANTIC 4 and 5 character editor in the December 1982 *BYTE* (reprinted in this book by permission of the author) and Charles Brannon's "SuperFont" from *COMPUTE!'s First Book of Atari Graphics*.

Any character editor can be used if you remember that instead of an 8-byte by 8-bit grid, each character is drawn on an 8-byte by *4-bit-pair* grid. A bit-pair of 00 selects the background color (register 4), and bit-pairs of 01, 10, and 11 select color registers 0, 1, and 2. Bit-pair 11, in inverse mode, selects register 3.

If the character set you ask for is not on the disk in drive 1, the program will prompt you either to insert the correct disk or to ask for a different set. Whenever Fontbyter asks you for a filename, you don't need to enter more than the eight-character name—Fontbyter always supplies the device name "D1:" and the extender ".SET" or ".SCR". If you use an illegal name, Fontbyter will ask you to try again.

Screen files. When you have chosen your character set, Fontbyter displays a directory of all the files with the filetype ".SCR". Fontbyter assumes that these files contain screen displays created and saved by Fontbyter. If no directory is displayed, it means that there are no files with the filetype ".SCR" on the disk.

At the end of the directory, you will be told the number of sectors left on the disk. Be sure that the disk you use for saving screens has enough room for the screen you intend to save. A maximum-size display is almost 4K, which will create a file of 33 sectors. Disks can fill up pretty fast at that rate.

Save file. The program asks you what name your saved screen file should have. When you are through editing and want to save your finished screen, this is the filename that Fontbyter will use to create the save file. You can use a filename that you used before, but saving the new file will erase the old one. Again, only the eight-letter filename is necessary. Fontbyter automatically selects "D1:" as the device name and ".SCR" as the filetype.

Load file. The program asks you if you want to edit a screen that was previously saved. If you do, you will be asked the name of the file you want to load from. (Again, only the eight-letter name will be used—the filetype must be ".SCR".) If the file is not found, Fontbyter will ask you either to enter another name or to insert the disk with that file on it; if you choose to enter another name, Fontbyter goes back to the original load file prompt, and you can decide at that point not to edit a previously saved file after all.

Notice that this sytem allows you to load from a file and then save your edited version back to the same file, erasing the old version; or you can choose to save the file under a different filename, so that both versions will exist. There is an added safeguard, too. When you save the screen display, it is first saved under the name "D1.TEMPFILE.SCR". Then Fontbyter asks you if you want to save it under the name you chose at the beginning of the program. If you change your mind about the save filename then, you can exit Fontbyter and use DOS to change "D:TEMPFILE.SCR" to whatever name you want.

Load file parameters. If your load file is found, Fontbyter immediately opens it and reads the first three bytes. Then it reminds you of the ANTIC mode, width (in characters), and height (in lines) of the file as it was saved. If you don't want to change those parameters, you can proceed directly to the final check; if you do want to change them, Fontbyter will ask you to choose the mode, width, and height of the file as if you were creating a new screen.

Changing the height or mode is fairly safe. Mode changes put twice as many lines on the screen, but all the relationships are the same. Changing the height merely adds or subtracts lines at the bottom of the display. Remember, though, that changing the width of a file will have very odd results. It does not cut off the edges of the old screen—it merely causes the lines to wrap around at a different point, so that nothing fits together vertically the way that it did.

ANTIC mode. Fontbyter asks you to choose which ANTIC mode you want. The only choices are 2 (GRAPHICS 0), 4, or 5. Mode 4 has shorter, squarer characters, and fits 24 lines on a screen. Mode 5 has tall, thin characters and fits only 12 lines on a screen. This means that a display file a hundred lines from top to bottom will give you more than eight distinct screen displays in ANTIC 5, but only just over four distinct displays in ANTIC 4. ANTIC 2 (GRAPHICS 0) is included, even though it is not a four-color mode, so that you can use Fontbyter to create displays using the built-in ROM character set.

Display width. The minimum width of a line is 40 characters. If you enter a number less than 40, Fontbyter will change it to 40. The maximum width depends on the mode. The limiting factor here is that all screen displays must fit within 4K. Because of this, the wider a screen display you choose, the fewer vertical lines you can have. You cannot have a line so wide that it would not allow the minimum number of lines. Since you will not be allowed any fewer than 24 screen lines in ANTIC 2 or 4, you naturally can't have as wide a screen as in mode 5, which has a minimum of 12 lines per screen.

Display height. The minimum height, in number of lines, is 12 lines for ANTIC 5 and 24 lines for ANTIC 2 and 4. The maximum height depends on the line width you chose. If you ask for more lines than the allowable maximum, Fontbyter will change the figure to the maximum.

Final check. Fontbyter clears the screen and then displays what your choices were: the character set, the file in which to save your screen, the file (if any) to load from, the mode, the width (in characters), and the height (in lines). If you want to make any changes, press OPTION. If you are satisfied with your choices, press START.

Fontbyter will display a wait message for a few moments, and then the screen will go completely blank. This is so the setup operations will run faster. When Fontbyter is ready to go on—and it won't be long—either the load screen you asked for will appear or a cursor will appear in the upper-left-hand corner of a blank screen. The cursor is whatever the ESCAPE character looks like in the character set you chose.

Also, part of the character set will be displayed on the bottom four lines of the screen. The characters are arranged in the same order as the computer keyboard, so that you can easily figure out which key to press in order to display a particular character.

Editing Features

To use the keyboard. The character set is divided into three groups: regular, shifted, and control. You can change from one to another using the CAPS/LOWR key. To get the regular character group, press CAPS/LOWR. To get the shifted character group, press SHIFT and CAPS/LOWR at the same time. To get the control character group, press CONTROL and CAPS/LOWR at the same time. As soon as you make the change, the character keyboard display at the bottom of the screen will change to show you the characters now available.

Instead of the usual computer keyboard system of locking only the alphabetic keys into shifted and control functions, Fontbyter shifts the entire keyboard. After you press SHIFT and CAPS/LOWR, you can press any key on the keyboard and get the shifted character—without pressing SHIFT again. The same applies to CONTROL with CAPS/LOWR.

Some keys, of course, don't have a shifted or control value (ESC, DEL, and RETURN, for instance), and others usually display only the inverse of another character (SHIFT-TAB, for instance). Since these don't display a separate character, pressing them only produces the same character that you would get if you pressed the space bar—a blank. (If your character set redefines the space bar character, that character will fill your display when it first comes up, and will appear on the screen whenever you enter a nonprinting character.)

The keys do not produce their normal clicking sound, except for the command keys, which are described next.

Command keys. No matter which character group you are using, there are some key combinations that Fontbyter interprets as commands. Pressing INSERT and SHIFT together will insert a blank line on the screen. Pressing DELETE and SHIFT together will delete a line. Pressing CONTROL and an ARROW key together will cause the cursor to move.

Remember, to print the *character* represented by the CONTROL-ARROW combination, press only the ARROW key while the control group is locked in. To move the cursor, press CONTROL and ARROW at the same time, regardless of which group is locked in.

Inverse video (Atari logo) key. This key is a toggle. Pressing it shifts you back and forth between inverse and regular video. In ANTIC 2 (GRAPHICS 0), this will cause all the characters you enter to be reversed, as the computer normally does. In ANTIC 4 and 5, however, this will cause Color 3 to take its value from color register 4 (memory location 711 instead of 710). It will affect, therefore, only one of the colors, and if a character does not contain any dots of Color 3, inverse mode won't have any effect at all.

CONTROL-ESC. This key combination is a toggle. Pressing it will shift you back and forth between Still and Auto-Advance modes. In Still Mode, pressing noncommand keys will display a new character in the same place on the screen. In Auto-Advance Mode, pressing noncommand keys will display a new character and then advance the cursor to the next position to the right, unless doing so would take the cursor beyond the edge of the display.

To move the cursor. Either move the joystick in the direction you want to move, or press the appropriate CONTROL-ARROW key combination. Only the joystick allows diagonal movement.

When the cursor reaches the edge of the screen, the display will begin to scroll until it reaches the limits of display height and width you specified during start-up. If you are at the edge of the display, the cursor simply won't move any farther that direction.

Fast-fill function. Sometimes you will have large areas or lines to fill with the same character. Instead of entering the character by typing it in each space where it is to appear, you can use the joystick and fire button. First maneuver the cursor until it is on top of the character you want to copy, or move it to the place where you want to begin the fast-fill operation and enter the character from the keyboard. Then press down the joystick button and hold it down while you use the joystick to move the cursor. From then on, until you let up on the button, wherever you move the cursor using the joystick, a trail made up of that character will be left behind.

You can also use this function to erase areas of the screen fairly quickly. Just move the cursor to a blank, press down the button, and the cursor will leave blanks behind it wherever you make it go.

Clear screen function. To erase the entire display, press CONTROL-SHIFT-CLEAR.

Delete line function. To delete an entire line of your screen, move your cursor to the line you want to delete and press SHIFT-DELETE. The line will vanish, and the entire display below that line will move upward one line on the screen. Whether the very bottom of your display is visible on the screen or not, a line of blanks will be inserted as the last line in your display.

Insert line function. To insert a blank line in your display, move the cursor to the position where you want the new line. Then press SHIFT-INSERT. The line that the cursor was on will move down, as will all the other lines below it in the display, and the cursor will now be on a blank line. At the bottom of the display, whether it is visible on the screen or not, the last line of your display will be deleted completely.

With both the delete and insert line functions, the line that disappears is irrecoverably lost. To get it back, you will have to enter all the characters just as you did before. So be careful about using these two functions!

By using the delete and insert functions in succession, you can quickly blank large areas of the screen, a line at a time. Simply move to the top of the area you want to blank out, and press SHIFT-DELETE as often as it takes to erase all the lines you wanted to get rid of. Then press SHIFT-INSERT until the desired number of blank lines appears.

You can also use these functions to move the entire picture upward or downward in the display. For instance, suppose you loaded a display that had been created and saved with only 24 lines, and you want to add another 24-line picture above it. At the beginning of the editing session, simply specify 48 lines as the height of the display. Fontbyter will put the 24 new blank lines at the end of the display. To move the old picture down into that blank area, start at the top of the screen and press SHIFT-INSERT 24 times.

Three joystick modes. We've already gone over the use of the joystick in Cursor Mode. The joystick can also be toggled into two other modes. If you press the START button while in Cursor Mode, the joystick will change to Scroll Mode. If you press the START button in Scroll Mode, the joystick will shift to Color Mode. And pressing the START button in Color Mode will shift you back to Cursor Mode again.

1. Scroll Mode. This mode enables you to scroll the TV screen

window around the entire display by moving the joystick in the appropriate direction. When you move, the cursor character will disappear. When you return to Cursor Mode, the cursor will come back to the middle of the screen.

2. Color Mode. In this mode, the joystick controls the color registers as follows:

- Forward and back: Color register 0 (Memory location 708)
- Left and right: Color register 1 (709)
- Forward and back with joystick button depressed: Color register 2 (710)
- Left and right with joystick button depressed: Background color register (712)
- Forward and back with SELECT depressed: Inverse color register (711)

As you press the joystick forward or to the right in Color Mode, that particular color will get brighter and brighter until it reaches maximum brightness; then it will jump to the next color at its darkest value and get brighter and brighter with that color. Pushing left or back cycles through the colors from bright to dark. There are 16 colors, each with eight levels of brightness. You can cycle through the colors endlessly in either direction.

When you start editing with a new display, the colors are the system default colors. When you load a previously saved display, however, you start with the colors saved with that display. You can change the colors however you like, and whatever the colors are when you save your display, those values will be saved with it.

Summary of Command Keys

Command Key	Description
START	Cycle from Cursor Mode to Scroll Mode to Color Mode and back to Cursor Mode.
SELECT	Save the current display without interrupting the edit. In Color Mode, moving the joystick forward and back with SELECT pressed will change the inverse color.
OPTION	Save the current display and stop the editing session.
CONTROL-ARROW	Move the cursor.
SHIFT-INSERT	Insert a blank line where the cursor is and delete the bottom line of the display.

SHIFT-DELETE	Delete the cursor line and add a blank line at the bottom of the display.
Atari logo key	Toggle back and forth between inverse and regular characters.
SHIFT-CAPS/LOWR	Select the shifted character group.
CONTROL-CAPS/ LOWR	Select the control character group.
CONTROL-ESC	Toggle between Still and Auto-Advance modes.
CONTROL-SHIFT-CLEAR	Erase the entire display.

Ending and Saving

There are two ways to save a screen.

1. You can press the SELECT button when the joystick is in Cursor Mode, and the display will be saved as "D1: TEMP-FILE.SCR". The screen is not changed, and you can resume editing as soon as the joystick or keyboard responds again.

2. You can press the OPTION button. Fontbyter will save the entire display in a file named "D1:TEMPFILE.SCR". The screen then clears, and Fontbyter asks if you want to save the display in the save file you asked for at the beginning of the edit. If you answer yes, "TEMPFILE" is renamed with the save filename you chose at the beginning. If a file with the same name already exists on the disk, it will be erased at this time.

If you are merely saving a half-done file to make sure some catastrophe doesn't lose it for you, then "TEMPFILE.SCR" should be security enough—if the system crashes, you'll know that the screen as you last saved it is in that file.

You will then be asked if you want to return to edit the same screen. If you say yes, your saved screen will quickly be reloaded into memory, and the program will reinitialize. If you say no, you will be asked whether you want to quit or start Fontbyter over again. If you choose the quit option and change your mind, don't worry. Just give the direct command RUN, and Fontbyter will begin again with the setup prompts.

Using Fontbyter Screens in Your Programs

Just because Fontbyter scrolls doesn't mean you have to make one continuous scrolling display. You can create many different screen displays in one file, "stacking" them vertically, and then use page flipping in your own program to move instantly from one to another. The advantage of using Fontbyter is that while you are creating the screen displays, you can scroll from one to the other to compare them and make sure that any animation effects you are trying for are working properly.

The diagrams will show you the variety of display configurations you can choose.

Figure 1 is the simplest and smallest display—just enter the minimum height and width for the mode you're working in.

Figure 2 is a completely horizontal display. You might want such a display in a game program to provide a very wide scramble-type game playfield, or in a storytelling program to allow the story to progress from left to right.

To get the widest possible display, use ANTIC 5. This allows displays that are 8.5 screens wide. However, the characters in ANTIC 5 are twice as tall as in ANTIC 4, which can distort your display.

If you want even wider displays, you can chain several displays together by beginning one display with the exact picture that ends the previous one; from there, continue the display as if there were no break. Then, when your own program scrolls to the end of one, it can page flip to the next 4K block of memory, where the next display begins, and continue scrolling. If you are doing horizontal scrolling, however, you are undoubtedly revising your display list in machine language, and the page flip will have to be in machine language, too. You can even make two displays chain to each other by having Display 1 begin with the end of Display 2, and Display 2 begin with the end of Display 1. That way the user keeps cycling through the same endless loop.

Figure 3 is a completely vertical display. You can use the same techniques to scroll vertically through a continuous display, except that now, since you are doing no horizontal scrolling, you can do scrolling and page flipping from BASIC just by POKEing new values into bytes 4 and 5 (counting from 0) of the display list.

In Figure 4, the screen is a window into a display that extends some distance both horizontally and vertically. You could use this in your programs to allow users to explore a map, or find their way out of a maze, or control hundreds of armies in a really massive and complex full-color war game, or simply to admire an elaborate picture.

Figure 1. The Minimum Display

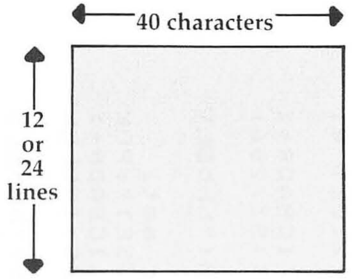


Figure 4. The Window Display

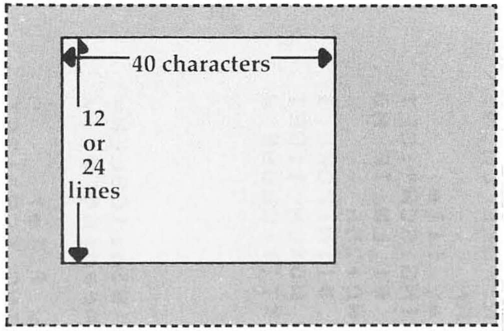


Figure 3. The Maximum Vertical Display

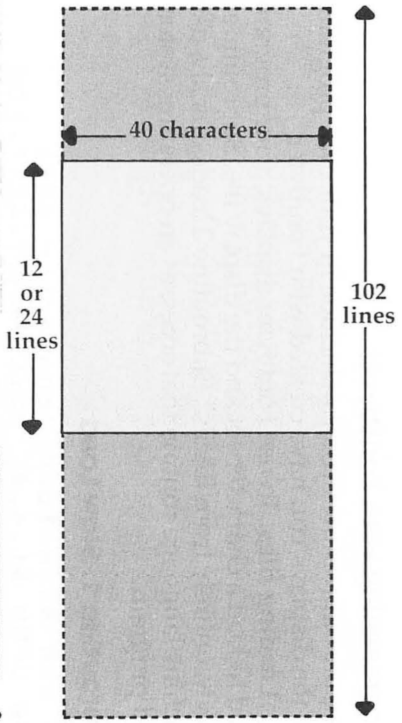
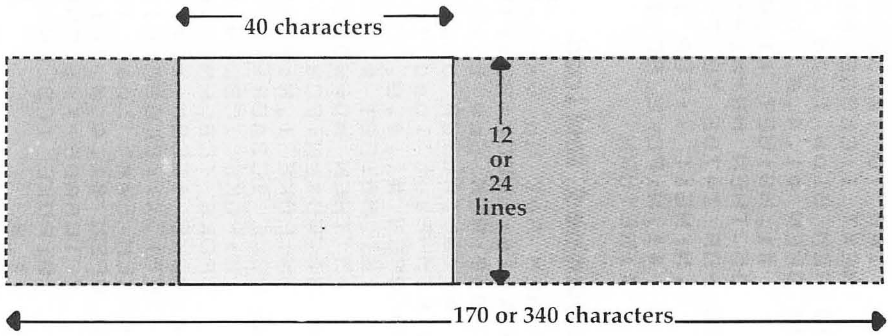


Figure 2. The Maximum Horizontal Display



Programming with Fontbyter

Here are subroutines you can include in your own programs to use the displays you have created with Fontbyter.

Loading files. To use Fontbyter displays, your program will need to load a character set and the display file. Subroutine 1 loads slowly, entirely from BASIC. Subroutine 2 loads quickly, using a machine language routine that accesses an operating system fast-load program.

Subroutine 1. Slow Load

```
1Ø REM Slow load (character set)
1ØØ OPEN #1,4,Ø,"D1:CHARACT.SET":FOR
     I=Ø TO 1Ø23:GET #1,N:POKE CH,N:
    NEXT I:CLOSE #1:RETURN
19Ø REM Slow load (display file)
2ØØ OPEN #1,4,Ø,"D1:DRAWING.SCR":GET
     #1,MD:GET #1,WD:GET #1,LN:IF MD
    >5 THEN MD=MD-1Ø:WD=WD+256
2Ø5 FOR I=7Ø8 TO 712:GET #1,N:POKE I
    ,N:NEXT I:FOR I=Ø TO WD*LN-1:GET
     #1,N:POKE SC+I,N:NEXT I:CLOSE #
    1:RETURN
```

Subroutine 2. Fast Load

```
1Ø REM Set up variables
2Ø X=16:ICCOM=834:ICBADR=836:ICBLEN=
   84Ø:REM See text for meaning of v
   ariables SP and CHBAS
9Ø REM Fast load (display file)
1ØØ OPEN #1,4,Ø,"D1:DRAWING.SCR":GET
     #1,MD:GET #1,WD:GET #1,LN:IF MD
    >5 THEN WD=WD+256:MD=MD-1Ø
11Ø SZ=WD*LN:FOR I=7Ø8 TO 712:GET #1
    ,N:POKE I,N:NEXT I
12Ø POKE ICBADR+X+1,SP:POKE ICBADR+X
    ,Ø:POKE ICBLEN+X+1,1+INT(SZ/256)
    :POKE ICBLEN+X,Ø
13Ø POKE ICCOM+X,7:I=USR(ADR("hhh▣LV
    ▣"),X):CLOSE #1:RETURN
19Ø REM Fast load (character set)
2ØØ OPEN #1,4,Ø,"D1:CHARACT.SET":POK
    E ICBADR+X+1,CHBAS:POKE ICBADR+X
    ,Ø:POKE ICBLEN+X+1,4:POKE ICBLEN
    +X,Ø
21Ø POKE ICCOM+X,7:C=USR(ADR("hhh▣LV
    ▣"),X):CLOSE #1:POKE 756,CHBAS:R
    ETURN
```

Display list setup. Subroutine 3 sets up an ANTIC 2 or 4 display list that can be horizontally or vertically scrolled. Subroutine 4 sets up an ANTIC 5 display list that can be horizontally or vertically scrolled. Subroutines 5 and 6 set up display lists that *cannot* be horizontally scrolled—use these only to load displays that were created with the minimum line width.

Subroutine 3. Horizontal Scroll Display List, ANTIC 2 or 4

```
10 REM Lines 20 and 30 are just a de
   monstration.  Change the value of
    SC and see what happens!
20 DL=PEEK(88)+256*PEEK(89):SC=DL:MO
   DE=4:WIDE=40:GOSUB 100
30 FOR I=0 TO 1000:NEXT I:SC=0:MODE=
   2:GOSUB 100:FOR I=0 TO 1000:NEXT
   I:GOTO 20
90 REM This ANTIC 2 or 4 display lis
   t can be horizontally scrolled.
   Just set the values of SC,DL,MODE
   , and WIDE.
100 FOR I=0 TO 2:POKE DL+I,112:NEXT
    I:N=0:M=MODE+64
110 FOR I=DL+3 TO DL+72 STEP 3:C=SC+
    N:POKE I,M:POKE I+2,INT(C/256):P
    OKE I+1,C-256*PEEK(I+2):N=N+WIDE
    :NEXT I
120 POKE I,65:DLHI=INT(DL/256):DLLO=
    DL-DLHI*256:POKE I+1,DLLO:POKE I
    +2,DLHI:POKE 561,DLHI:POKE 560,D
    LLO:RETURN
```

Subroutine 4. Horizontal Scroll Display List, ANTIC 5

```
10 REM Lines 20 and 30 are just a de
   monstration.  Change the value of
    SC and see what happens!
20 DL=PEEK(88)+256*PEEK(89):SC=PEEK(
   106)*256:MODE=5:WIDE=40:GOSUB 100
30 FOR I=0 TO 1000:NEXT I:SC=0:GOSUB
    100:FOR I=0 TO 1000:NEXT I:GOTO
   20
90 REM This ANTIC 5 display list can
    be horizontally scrolled.  Just
   set the values of SC,DL,MODE, and
    WIDE.
100 FOR I=0 TO 2:POKE DL+I,112:NEXT
    I:N=0:M=MODE+64
```

```
110 FOR I=DL+3 TO DL+36 STEP 3:C=SC+
    N:POKE I,M:POKE I+2,INT(C/256):P
    OKE I+1,C-256*PEEK(I+2):N=N+WIDE
    :NEXT I
120 POKE I,65:DLHI=INT(DL/256):DLLO=
    DL-DLHI*256:POKE I+1,DLLO:POKE I
    +2,DLHI:POKE 561,DLHI:POKE 560,D
    LLO:RETURN
```

Subroutine 5. Regular Display List, ANTIC 2 or 4

```
10 REM The actual subroutine is line
   s 100-120.  You set the value of
   DL,SC,MODE,and WIDE.
20 DL=PEEK(88)+256*PEEK(89):MODE=2:W
   IDE=40
30 SC=0:MODE=2+2*(MODE=2):GOSUB 100
40 TRAP 30:ON PEEK(753)<>3 GOTO 40:S
   C=SC+480:SP=INT(SC/256):POKE DL5,
   SP:POKE DL4,SC-256*SP
50 FOR I=0 TO 30:NEXT I:GOTO 40
90 REM This ANTIC 2 and 4 display li
   st can be page flipped from BASIC
   .  POKE the screen address into D
   L4 and DL5.
100 FOR I=0 TO 2:POKE DL+I,112:NEXT
    I:DL4=DL+4:DL5=DL+5
110 POKE DL+3,64+MODE:POKE DL5,INT(S
    C/256):POKE DL4,SC-256*PEEK(DL5)
    :FOR I=DL+6 TO DL+28:POKE I,MODE
    :NEXT I
120 POKE I,65:DLHI=INT(DL/256):DLLO=
    DL-DLHI*256:POKE I+1,DLLO:POKE I
    +2,DLHI:POKE 561,DLHI:POKE 560,D
    LLO:RETURN
```

Subroutine 6. Regular Display List, ANTIC 5

```
10 REM The actual subroutine is line
   s 100-120.  You set the value of
   DL,SC,MODE,and WIDE.
20 DL=PEEK(88)+256*PEEK(89):MODE=5:W
   IDE=40:GOSUB 100
30 SC=0
40 TRAP 30:ON PEEK(753)<>3 GOTO 40:S
   C=SC+480:SP=INT(SC/256):POKE DL5,
   SP:POKE DL4,SC-256*SP
50 FOR I=0 TO 30:NEXT I:GOTO 40
```

```
90 REM This ANTIC 5 display list can
    be page flipped from BASIC.  Jus
   t POKE the screen address into DL
   4 and DL5.
100 FOR I=Ø TO 2:POKE DL+I,112:NEXT
    I:DL4=DL+4:DL5=DL+5
110 POKE DL+3,64+MODE:POKE DL5,INT(S
    C/256):POKE DL4,SC-256*PEEK(DL5)
    :FOR I=DL+6 TO DL+16:POKE I,MODE
    :NEXT I
120 POKE I,65:DLHI=INT(DL/256):DLLO=
    DL-DLHI*256:POKE I+1,DLLO:POKE I
    +2,DLHI:POKE 561,DLHI:POKE 560,D
    LLO:RETURN
```

These routines use the following variables:

TOP is the page number of the top of memory. The Atari will not touch anything located above the top of memory—but anything below it is fair game. The display list, character set, screen memory, and machine language routines should all be placed above SP. So the load routines find out where the top of memory is and move it down enough pages to leave room for all the protected program areas. SC is the absolute address of the top of memory (SP*256); it is also the start of screen memory, so that it is POKEd into both the display list and location 106.

How much room should you leave? The character set takes 1K (four pages) and must start on a 1K boundary. Screen memory will never take more than 4K (16 pages), and should start on a 4K boundary, since ANTIC has problems when screen memory crosses that line. If your display is less than 2K, you can probably skip back from the top of memory a mere 4K (16 pages, or PEEK(106)-16), place screen memory at the new top of memory, and put the display list, machine language routines, and character set above it. If your display list is 3K or more, you should probably skip back 6K (24 pages, or PEEK(106)-24), place the character set at the new top of memory, followed by the display list, machine language subroutines, and then screen memory beginning at the 4K boundary line, 16 pages before the old top of memory. This routine assumes that arrangement.

SP is the page number of the start of screen memory, and SC is the absolute address of the start of screen memory (SP*256).

DL is the start of the display list. For page flipping, DL3 is DL + 3, and DL4 is DL + 4. These will contain the low byte and high byte of screen memory, and POKEing new values into these locations will flip screen memory.

CHBAS is the page number of the character set, and CH is the absolute address (CHBAS*256).

MODE is the ANTIC mode number—either 2, 4, or 5. Adding 64 to MODE each time it is POKEd in tells the computer to look for a new screen memory address in the next two bytes in the display list.

WIDE is the width, in characters, of the entire horizontal line, not just the 40-character portion visible on screen at any one time. Thus, every MODE instruction is followed by a two-byte address, C, which tells it where to find the start of the next horizontal line.

POKEing 560 and 561 with 0 and DL/256 is what actually makes the display list start working. Until that moment, the display list is just a series of numbers in memory. But once 560 and 561 contain the address of the start of your display list, the TV screen is under your program's control.

ICBADR, ICBLEN, and ICCOM are the addresses of key locations in the IOCB handler. ICCOM must contain the number of the operation to be performed (7 to load, 11 to save). ICBADR must contain the low byte of the starting address of the area in memory to be saved from or loaded to (ICBADR + 1 will contain the high byte). ICBLEN must contain the low byte of the length of the file to load (ICBLEN + 1 will contain the high byte). The variable X represents the offset into the IOCB area. If you OPEN #1, then X = 16. If you OPEN #2, then X = 32. And so on, in multiples of 16. You might not get good results using OPEN #0 or OPEN #6—those are reserved for system use.

With screen files created by Fontbyter, remember that the first eight bytes always contain the following information:

- ANTIC mode number (plus 10, if width is greater than 255 characters)
- width, or number of characters per line (low byte only, if width is greater than 255 characters)
- display height, or number of lines in the entire display
- colors to POKE into locations 708 through 712

To calculate the number of bytes in the whole screen display (SZ), multiply the height by the width. The number of bytes in the file is that number plus eight.

Typing the Program

The bulk of the program is written in BASIC. The shortest machine language routines are included as string constants. The longer routines, however, DISPLAY, EXPAND, and DELETE, and two data files, MENU.DAT and CHARDATA.DAT, are listed after the main program. These should be entered using the BASIC loader program provided and saved on disk with exactly the filename specified. Fontbyter will look for these files and load them into strings or particular areas of memory during the run of the program.

Since Fontbyter works most efficiently with a disk drive, the program as written assumes a disk drive. However, a patient cassette user can remove all the routines related to choosing and testing filenames, and simply assign the value "C:" to all filename variables. You may also want to add prompts to tell the user what file the program is asking for. (The biggest problem arises with load files during initialization, when the program tests the saved screen file once, then loads it again later. If you decide not to revise the program, make sure that you rewind the cassette containing the screen file after that initial test, so the file will be complete when it is loaded by the screen load subroutine.)

There are no REM statements in Fontbyter. Instead, remarks are included as typeset lines that interrupt the program listing. Type only the numbered program lines that were printed out by a printer, not the comments printed in the same typeface you are reading now.

Program 1. Fontbyter

Start-up. DIMension arrays and string variables; clear the screen; establish key addresses.

Line 5. F$ is the character set name; FSAVE$ is the save filename; FLOAD$ is the load filename; FL$ and FLL$ are temporary filenames for various uses; DELETE$ and EXPAND$ will hold machine language subroutines in ASCII form; C(*n*) will hold the relationship between keyboard code and internal code.

10. Clear the screen. ICCOM, ICBADR, and ICBLEN are addresses in Input/Output Control Block (IOCB) zero (for files opened as #0). ICCOM is the address where the command should be POKEd; ICBADR should be POKEd with the low byte of the address in memory where files should be stored or loaded (ICBADR+1 holds the high byte); ICBLEN should be POKEd with the low byte of the length of the file to be moved (ICBLEN+1 holds the high byte). X is the offset into succeeding IOCBs. To use file #1, X=16; for file #2, X=32; and so on.

15. COL1 through COL5 are the addresses of the color registers. COL4 is the inverse mode color; COL5 is the background color. SHIF keeps track of which character mode we are locked into—unshifted (0), shifted (64), control (128), or shift-control (192). SCON is the value to turn the screen back on. POKE 16, 112 disables the BREAK key. Jump to 440.

```
5 DIM F$(20),FSAVE$(20),FLOAD$(20),F
  L$(40),FLL$(20),DELETE$(124),EXPAN
  D$(124),CLEAR$(33),C(255)
10 GRAPHICS 0:X=16:ICCOM=834:ICBADR=
   836:ICBLEN=840
15 COL1=708:COL2=709:COL3=710:COL4=7
   11:COL5=712:SHIF=64:SCON=PEEK(559
   ):POKE 16,112:GOTO 440
```

Display Load Subroutine. MD is the mode number; WD is the number of characters per line; LN is the number of lines in the display. If the width of the screen is greater than 255, the high byte of the width has been stored in MD by adding 10 to the mode number. The size of the display is (adjusted) WD*LN. The colors are read; then ICBADR is POKEd with the starting address of screen memory; ICBLEN is POKEd with the length of the remainder of the file; and ICCOM is POKEd with 7, which is the read-from-disk command. The machine language subroutine merely pulls three numbers off the stack and jumps to a subroutine built into the operating system.

```
20 OPEN #1,4,0,FL$:GET #1,MD:GET #1,
   WD:GET #1,LN:IF MD>5 THEN WD=WD+2
   56:MD=MD-10
25 SZ=WD*LN:FOR I=COL1 TO COL5:GET #
   1,N:POKE I,N:NEXT I
30 SC=SP*256:POKE ICBADR+X+1,SP:POKE
    ICBADR+X,0:POKE ICBLEN+X+1,1+INT
   (SZ/256):POKE ICBLEN+X,0
35 POKE ICCOM+X,7:I=USR(ADR("hhh⊞LV◩
   "),X):CLOSE #1:RETURN
```

Display Save Subroutine. The Display Load Subroutine in reverse. ICCOM is now POKEd with 11, the write-to-disk command. This is where the program adds 10 to MD if WD is greater than 255. The filename is always "D1:TEMPFILE.SCR".

```
40 OPEN #1,8,0,"D1:TEMPFILE.SCR":WD=
   WIDE:MD=MODE:IF WIDE>255 THEN WD=
   WIDE-256:MD=MODE+10
```

```
45 PUT #1,MD:PUT #1,WD:PUT #1,LINE:F
   OR I=COL1 TO COL5:PUT #1,PEEK(I):
   NEXT I
5Ø POKE ICBADR+X+1,SP:POKE ICBADR+X,
   Ø:POKE ICBLEN+X+1,1+INT((LINE*WID
   E)/256):POKE ICBLEN+X,Ø
55 POKE ICCOM+X,11:I=USR(ADR("hhh▩LV
   ▩"),X):CLOSE #1:RETURN
```

Delete Line Subroutine. Line 60 quits the routine if the cursor is on the bottom line of the display. LOWAD is the address of the leftmost character on the cursor line; HIADD is the line below it on the screen (which is *above* it in memory). LOWAD is POKEd into 203-204, and HIADD into 205-206. Address 207 is POKEd with the number of lines between the cursor line and the end of the display, and 208 and 209 hold the low and high bytes of the width of a line. The machine language subroutine in the string DELETE$ moves everything below the cursor line up one line on the screen, and inserts a blank line at the bottom of the display. The cursor (91 = ESC) is then POKEd back into place.

```
6Ø IF ((LINE*WIDE-PIX)<WIDE) THEN RE
   TURN
65 LOWAD=SC+WIDE*INT(PIX/WIDE)-1:HIA
   DD=LOWAD+WIDE:POKE 2Ø6,INT(HIADD/
   256):POKE 2Ø5,HIADD-PEEK(2Ø6)*256
7Ø POKE 2Ø4,INT(LOWAD/256):POKE 2Ø3,
   LOWAD-PEEK(2Ø4)*256:POKE SC+PIX,O
   LD
75 POKE 2Ø7,INT((LINE*WIDE-PIX)/WIDE
   ):POKE 2Ø8,WLO:POKE 2Ø9,WHI
8Ø C=USR(ADR(DELETE$))
85 OLD=PEEK(SC+PIX):POKE SC+PIX,91:R
   ETURN
```

Insert Line Subroutine. The same as the Delete Line Subroutine, except that everything from the cursor line to the end of the display is moved down the screen, and a blank line is inserted at the cursor line. LOWAD is now the second to last line of the display, and HIADD is the last line of the display.

```
9Ø IF ((LINE*WIDE-PIX)<WIDE) THEN RE
   TURN
95 HIADD=SC+WIDE*(LINE-1)-1:LOWAD=HI
   ADD-WIDE:POKE 2Ø6,INT(HIADD/256):
   POKE 2Ø5,HIADD-PEEK(2Ø6)*256
1ØØ POKE 2Ø4,INT(LOWAD/256):POKE 2Ø3
    ,LOWAD-PEEK(2Ø4)*256:POKE SC+PIX,OLD
```

```
105 POKE 207,INT((LINE*WIDE-PIX)/WID
    E):POKE 208,WLO:POKE 209,WHI
110 C=USR(ADR(EXPAND$))
115 OLD=0:POKE SC+PIX,91:RETURN
```

Main Loop. Most of the time, the program cycles through from line 125 to line 150; unless the user presses a key or moves the joystick, this will go on endlessly.

120. Store the value of whatever character belongs in screen memory where the cursor now is in the variable OLD. When the cursor moves off, OLD will be POKEd back in. The internal code for the ESC character is 91—it is used as the cursor. POKE 559,SCON turns the screen on. This line is executed only once.

125. Zero out several variables and check for user input. OPT holds the value returned by the console keys, DI the value returned by joystick 1, and T the value of fire button 1.

130. If the START button is pressed, jump to the Quick Scroll Subroutine at 870; when that is over, jump to the Color Change Subroutine at 255.

135. If the joystick has been pushed, jump to the Cursor Movement Subroutine at 155.

140. If a key has been pressed (PEEK(753) = 3), jump to the Read Keyboard Subroutine at 220. Upon RETURNing, the flag MV will tell whether the arrow keys or auto-advance mode were used to move the cursor; if yes (MV = 1), jump to the third line of the Cursor Movement Subroutine at 165.

145. If OPTION has been pressed, jump to the Stop Edit Subroutine at 705. If SELECT has been pressed, jump to the Display Save Subroutine. The display will be saved as "D1:TEMPFILE.SCR" without interrupting the edit.

150. Start the loop over again.

```
120 OLD=PEEK(SC+PIX):POKE SC+PIX,91:
    POKE 559,SCON:POKE 16,112
125 MV=0:V=0:H=0:OPT=PEEK(53279):DI=
    PEEK(632):T=PEEK(644):E=0
130 IF OPT=6 THEN GOSUB 870:GOSUB 26
    0:GOTO 125
135 IF DI<15 THEN GOSUB 155:GOTO 125
140 IF PEEK(753)=3 THEN GOSUB 220:ON
     MV GOSUB 165:GOTO 125
145 ON OPT=3 GOTO 705:IF OPT=5 THEN
    GOSUB 40:POKE SC+PIX,91:GOTO 125
150 GOTO 125
```

Cursor Movement Subroutine. In this routine, V is vertical movement; H is horizontal movement; PIX is the current position of the cursor relative to the start of screen memory; OLD is the character "under" the cursor in screen memory; UD is the current cursor line; LR is the current left-right position of the cursor on a line; W is the position of the upper-left-hand corner of the current screen display relative to the start of screen memory; U is the current line number of W; and L is the current left-right position of W on its line. WH is the horizontal scroll flag (1 = left, 2 = right); WV is the vertical scroll flag (1 = up, 2 = down).

155-160. Convert the joystick position into cursor move instructions.

165-175. Get the current cursor position. If the called-for movement would take the cursor beyond the edge of the display, cancel that movement instruction. If no movement is called for, quit the subroutine. Keyboard-controlled cursor movements enter the routine here.

180-195. Get the current screen position relative to the start of screen memory. If scrolling is necessary, execute the machine language scroll subroutine at address DISPLAY.

200. Put the OLD character into PIX; change PIX to its new value; sound the console buzzer (POKE 53279, 1).

205-215. If the trigger is not pressed, put the character at PIX in the variable OLD and POKE the cursor character into the new cursor position.

```
155 V=WIDE*((DI=9 OR DI=13 OR DI=5)-
    (DI=10 OR DI=14 OR DI=6)):POKE 7
    7,0
160 H=(DI=6 OR DI=7 OR DI=5)-(DI=10
    OR DI=11 OR DI=9)
165 UD=INT(PIX/WIDE):IF UD-(V<0)<0 O
    R UD+(V>0)=LINE THEN V=0
170 LR=PIX-WIDE*UD:IF LR+H<0 OR LR+H
    >WIDE-1 THEN H=0
175 IF H=0 AND V=0 THEN 215
180 WH=0:WV=0:W=PEEK(DL4)+256*PEEK(D
    L5)-SC
185 U=INT(W/WIDE):IF V<>0 THEN WV=(U
    D-U-(V<0)<0)+2*(UD-U+(V>0)>8+12*
    (MODE<>5))
190 IF H<>0 THEN L=W-U*WIDE:WH=(LR+H
    -L<0)+2*(LR+H-L>39)
195 IF WH>0 OR WV>0 THEN POKE DL+114
    ,WH:POKE DL+115,WV:C=USR(DISPLAY
    )
```

```
200 POKE SC+PIX,OLD:PIX=PIX+H+V:POKE
     53279,1
205 IF T=1 THEN OLD=PEEK(SC+PIX):POK
    E SC+PIX,91:GOTO 215
210 POKE SC+PIX,OLD
215 RETURN
```

Read Keyboard Subroutine. C is the actual value of the key combination pressed (key plus SHIFT and/or CONTROL); N is the absolute value of the key depressed (key minus SHIFT and/or CONTROL); SHIF is the offset for unshifted, shifted, control, or shift-control character groups (0 = unshifted, 64 = shifted, 128 = control, and 192 = shift-control); VERS is the high-bit condition for inverse mode (regular = 0, inverse = 128); and MV is the flag for whether movement is to take place upon leaving the subroutine (1 = movement).

220. Get the key pressed from the Keycode Subroutine at 785; if the key is an arrow key, jump to the arrow key routine at 250; if it is SHIFT-DELETE, jump to the Delete Line Subroutine at 60; if it is SHIFT-INSERT, jump to the Insert Line Subroutine at 90.

225. If the key was CONTROL-ESC, toggle the auto-advance flag AV from 0 to 1 or back again.

230. If the key was CAPS-LOWR, set SHIF to the appropriate value. CAPS-LOWR alone = 0, SHIFT-CAPS-LOWR = 64, CONTROL-CAPS-LOWR = 128, and SHIFT-CONTROL-CAPS-LOWR = 192. Jump to the subroutine at 930, which changes the keyboard display on the bottom four lines of the screen to show the current character group.

235. If the key was the Atari logo key (or inverse key), toggle the value of VERS from 0 to 128 or back again.

240-245. If none of the above commands was executed, change OLD to the internal character code of the character called for, using the array C(*n*), and POKE that value into the current cursor position (SC + PIX). If AV is set to 1, change the value of C to 135 (right arrow).

250. Set the value of V and H as required by the arrow key pressed, and set the flag MV to 1 to indicate that a move is called for.

```
220 GOSUB 785:ON (C=134)+(C=135)+(C=
    142)+(C=143)+2*(C=116)+3*(C=119)
    +4*(C=246) GOTO 250,60,90,645
225 IF C=156 THEN AV=1*(AV=0):GOTO 9
    20
```

```
230 IF N=60 THEN SHIF=4+C-64:POKE 53
    279,4:GOSUB 930:RETURN
235 IF N=39 THEN VERS=128*(VERS=0):G
    OTO 920
240 OLD=C(N+SHIF)+VERS:POKE SC+PIX,O
    LD:ON AV GOTO 245:RETURN
245 C=135
250 V=WIDE*((C=143)-(C=142)):H=(C=13
    5)-(C=134):MV=1:RETURN
```

Color Change Subroutine. DI is the joystick position, T is the joystick trigger (0 = pressed, 1 = not pressed), OPT is the console key, and COL1-COL5 are the addresses of the color registers.

255-260. The exit and entrance routines. 920 is the Delay Subroutine. The exit routine also POKEs the cursor character into place and RETURNs.

265-275. Read the joystick, button, and console keys. If START is pressed, exit the routine through 255. The four legal directions are 7, 11, 13, and 14. Change a 7 to a 12, then subtract 10; now if the joystick is indicating a legal direction, the value of DI will be 1, 2, 3, or 4. If it isn't one of those, go back and read again. If it is, use those plus T to get to the right line.

280-315. Read the color register and change the value as appropriate. Lines 310 and 320 change the inverse color register (COL4) if SELECT was pressed; otherwise, line 315 or 325 is executed.

```
255 GOSUB 920:POKE SC+PIX,91:RETURN
260 GOSUB 920
265 DI=PEEK(632):T=PEEK(644):DI=DI+5
    *(DI=7):DI=DI-10:OPT=PEEK(53279)
    :IV=(OPT=5):IF OPT=6 THEN 255
270 IF DI<1 OR DI>4 THEN 265
275 ON (4*T)+DI GOSUB 280,285,290,29
    5,300,305,310,320:GOTO 265
280 POKE COL5,PEEK(COL5)-2+256*(PEEK
    (COL5)<2):RETURN
285 POKE COL5,PEEK(COL5)+2-256*(PEEK
    (COL5)>253):RETURN
290 POKE COL3,PEEK(COL3)-2+256*(PEEK
    (COL3)<2):RETURN
295 POKE COL3,PEEK(COL3)+2-256*(PEEK
    (COL3)>253):RETURN
300 POKE COL2,PEEK(COL2)-2+256*(PEEK
    (COL2)<2):RETURN
305 POKE COL2,PEEK(COL2)+2-256*(PEEK
    (COL2)>253):RETURN
```

```
310 IF IV THEN POKE COL4,PEEK(COL4)-
    2+256*(PEEK(COL4)<2):RETURN
315 POKE COL1,PEEK(COL1)-2+256*(PEEK
    (COL1)<2):RETURN
320 IF IV THEN POKE COL4,PEEK(COL4)+
    2-256*(PEEK(COL4)>253):RETURN
325 POKE COL1,PEEK(COL1)+2-256*(PEEK
    (COL1)>253):RETURN
```

Filename Test Subroutine. This routine takes apart the filename that the user entered; removes anything before a colon, anything after a period, and any letters beyond the first eight; and provides error messages if there is nothing left, or if the name doesn't begin with a capital letter, or if there are illegal characters in the filename. If the name failed, flag *N* is set to 1.

```
330 FLL$=FL$:FOR I=1 TO LEN(FL$):N=A
    SC(FL$(I,I)):ON N=58 GOSUB 370:N
    EXT I:FL$=FLL$
335 FLL$=FL$:FOR I=1 TO LEN(FL$):N=A
    SC(FL$(I,I)):ON N=46 GOSUB 375:N
    EXT I:FL$=FLL$
340 IF LEN(FL$)>8 THEN FL$=FL$(1,8)
345 IF LEN(FL$)<1 THEN 390
350 N=ASC(FL$(1,1)):IF N>90 OR N<65
    THEN 385
355 IF LEN(FL$)<2 THEN GOTO 365
360 FOR I=2 TO LEN(FL$):N=ASC(FL$(I,
    I)):ON (N>90 OR N<65) AND (N>57
    OR N<48) GOTO 380:NEXT I
365 FLL$="D1:":FLL$(4)=FL$:N=0:RETUR
    N
370 FLL$=FL$(I+1,LEN(FL$)):RETURN
375 FLL$=FL$(1,I-1):RETURN
380 POP :? "{CLEAR}":? "Illegal char
    acters in ";FL$:GOTO 390
385 ? "{CLEAR}":? FL$;" must start w
    ith a capital":? "letter.":GOTO
    390
390 ? "Let's try that name again.":N
    =1:RETURN
```

Disk Test Subroutines. These routines test to see if a file is present on the disk. If not, flag *N* is set to 1.

395-410. Check to see if file FL$ is on the disk. If it *isn't*, warn the user.

415-435. Check to see if file FL$ is on the disk. If it *is*, warn the user.

```
395 TRAP 400:OPEN #1,4,0,FL$:N=0:CLO
    SE #1:RETURN
400 ? :? FL$;" isn't on disk in":? "
    drive 1":? "Insert disk with ";F
    L$;"and":? "press RETURN.":CLOSE
     #1
405 ? "Or to try another file name,
    press anyother key."
410 ON PEEK(753)<>3 GOTO 410:GOSUB 7
    85:ON N=12 GOTO 395:N=1:RETURN
415 TRAP 435:OPEN #1,4,0,FL$:? FL$;"
     is already on disk.":? "Unless
    you change the name, the old"
420 ? "file will be lost.  To change
     the namepress RETURN":? "Or pre
    ss any other key to continue.":C
    LOSE #1
425 ON PEEK(753)<>3 GOTO 425:GOSUB 7
    85:ON N=12 GOTO 430:N=0:RETURN
430 N=1:RETURN
435 CLOSE #1:N=0:RETURN
```

Setup Routine. This long routine sets up the parameters for editing or creating the display.

440. Print the program name and run the Keycode Array Subroutine at 905.

445-460. Get a character set directory through the Directory Subroutine at 850, then get the user's choice of character set and test it through the Filename Test Subroutine at 330 and the Disk Test Subroutine at 395.

465-475. Get a screen file directory through the Directory Subroutine at 840, then get the user's save filename (FSAVE$) and test it at 330 and 415.

480-485. Find out if the user wants to edit a previously saved file. If not, skip on to 535.

490-530. Get the user's load filename and test it at 330 and 395. If the filename fails, go back to 480 so the user has the option of not loading a file after all. If the filename succeeds, then get the mode (MD), width (WD), and length (LN) parameters from the load file. Ask if the user wants to change the parameters. If not, jump to 585 for the final check; if yes, then proceed to 540.

535. Set the flag FLOAD to 0, which means there is no load file.

540-550. Get the user's choice of mode. Only 2, 4 and 5 are acceptable.

555-560. Get the user's choice of line width (WIDE). The minimum is 40 characters per line; the subroutine at 790 determines the maximum.

565-580. Get the user's choice of number of lines in the display (LINE). Minimum and maximum depend on mode and line width. The display cannot be longer than 4K.

585-600. Show the user what the parameters are and get the user's decision about whether to proceed or not. If not, go back to 440.

```
440 ? "{13 SPACES}Fontbyter":? :? :?
    :GOSUB 905
445 GOSUB 850:? "What is the name of
     your character{4 SPACES}set? (E
    nter '@' for ROM set)":POKE 764,
    255:INPUT F$
450 IF F$="@" THEN 465
455 FL$=F$:GOSUB 330:ON N GOTO 445:F
    $=FLL$:F$(LEN(FLL$)+1)=".SET"
460 FL$=F$:GOSUB 395:ON N GOTO 445
465 GOSUB 840:? :? "What file should
     hold your finished{3 SPACES}scr
    een? (Eight characters)":POKE 76
    4,255:INPUT FSAVE$
470 FL$=FSAVE$:GOSUB 330:ON N GOTO 4
    65:FSAVE$=FLL$:FSAVE$(LEN(FLL$)+
    1)=".SCR"
475 FL$=FSAVE$:GOSUB 415:ON N GOTO 4
    65
480 FLOAD$="":? :? "Would you like t
    o edit a screen you{3 SPACES}hav
    e already saved? (Y or N) "
485 GOSUB 785:ON N=35 GOTO 535:ON N=
    43 GOTO 490:GOTO 485
490 ? :? "What is the name of the sa
    ved screen  file? ":POKE 764,255
    :INPUT FLOAD$
495 FL$=FLOAD$:GOSUB 330:ON N=0 GOTO
     500:GOTO 480
500 FLOAD$=FLL$:FLOAD$(LEN(FLL$)+1)=
    ".SCR"
505 FL$=FLOAD$:GOSUB 395:ON N GOTO 4
    80:OPEN #1,4,0,FLOAD$:GET #1,MD:
    GET #1,WD:GET #1,LN:CLOSE #1:FLO
    AD=1
510 IF MD>5 THEN MD=MD-10:WD=WD+256
515 ? :? FLOAD$;" was saved as:":? "
    Mode ";MD;",":? "with ";LN;" lin
    es":? "of ";WD;" characters per line."
```

```
520 ? "If you wish to change these p
    arameterspress RETURN.":? "To le
    ave them unchanged press any
    {5 SPACES}other key."
525 ON PEEK(753)<>3 GOTO 525:GOSUB 7
    85:IF N=12 THEN 540
530 MODE=MD:WIDE=WD:LINE=LN:GOTO 585
535 FLOAD=0
540 ? :? "What Antic mode will you w
    ork in?":? "(Antic 2, 4, OR 5) "
    :POKE 764,255
545 GOSUB 785:ON N<>30 AND N<>24 AND
     N<>29 GOTO 545
550 MODE=C(N)-16
555 ? :? "How wide a line?":? "  (Mi
    nimum 40 characters":? "
    {3 SPACES}maximum ";170+170*(MOD
    E=5);" characters)"
560 POKE 764,255:TRAP 560:INPUT WIDE
    :WIDE=INT(WIDE):ON WIDE<40 OR WI
    DE>170 GOSUB 790
565 ? :? "How many lines do you want
     to edit?{5 SPACES}(Minimum ";12
    +12*(MODE<>5);:? "{3 SPACES}Maxi
    mum ";INT(4096/WIDE);")"
570 TRAP 570:INPUT LINE
575 LINE=INT(LINE):IF LINE>INT(4096/
    WIDE) THEN LINE=INT(4096/WIDE)
580 IF LINE<12+12*(MODE=4) THEN LINE
    =12+12*(MODE=4)
585 ? "{CLEAR}":? "You have chosen:"
    :? "Character set--";F$:? "Save
    file--";FSAVE$:? "Load file--";F
    LOAD$
590 SZ=LINE*WIDE-1:? "Mode ";MODE:?
    LINE;" lines of ";WIDE;" charact
    ers"
595 ? "If this is right, press START
    {9 SPACES}To make changes, press
     OPTION"
600 ON (PEEK(53279)=6)+(2*(PEEK(5327
    9)=3)) GOTO 605,440:GOTO 600
```

Initialization. A is the old top of memory; TOP is the new top of memory; CHBAS is the page number of the character set; CH is the absolute address of the character set; SP is the page number of the start of screen memory; SC is the absolute address of the start of screen memory; and OLDCHBAS is the ROM character set page number. (ICCOM, ICBADR, and ICBLEN are explained in the notes for lines 20-35.)

605. Move memory down by 24 pages (6K) to reserve space for screen memory, the character set, the display list, and other protected data. Set addresses.

610-615. Print the wait message. If the user specified the ROM character set ("@"), skip to 630.

620-625. Use a machine language routine to read the character set from disk and load it at CH.

630. Turn off the screen, run the setup subroutines, and jump to the main loop at 120.

```
605 A=PEEK(1Ø6):TOP=A-24:CHBAS=TOP:C
    H=CHBAS*256:SP=TOP+8:SC=SP*256:P
    OKE 1Ø6,TOP:OLDCHBAS=224:GRAPHIC
    S Ø
61Ø ? "Just a minute while I get mys
    elf{6 SPACES}together . . ."
615 IF F$="@" THEN CHBAS=224:CH=CHBA
    S*256:GOTO 63Ø
62Ø OPEN #1,4,Ø,F$:POKE ICBADR+X+1,C
    HBAS:POKE ICBADR+X,Ø:POKE ICBLEN
    +X+1,4:POKE ICBLEN+X,Ø
625 POKE ICCOM+X,7:C=USR(ADR("hhh▩LV
    ▩"),X):CLOSE #1
63Ø POKE 559,Ø:GOSUB 64Ø:GOSUB 655:G
    OSUB 81Ø:GOSUB 635:ON FLOAD GOSU
    B 65Ø:GOSUB 925:GOTO 12Ø
```

Set CHBAS Subroutine. Tell the operating system where the character set is.

```
635 POKE 756,CHBAS:RETURN
```

Clear Memory Subroutine. Machine language routine to fill screen memory with zeros. (Anything in brackets should be entered with the CONTROL key depressed.)

```
64Ø OPEN #1,4,Ø,"D1:CLEAR.SUB":FOR I
    =1 TO 33:GET #1,N:CLEAR$(I,I)=CH
    R$(N):NEXT I:CLOSE #1
645 C=USR(ADR(CLEAR$),SP,X):RETURN
```

Load FLOAD$ Subroutine. Set FL$ TO FLOAD$ and run the Display Load Subroutine at 20.

```
65Ø T=SZ:FL$=FLOAD$:GOSUB 2Ø:SZ=T:RE
    TURN
```

Display List Subroutine. DL is the address of the display list; DL4 + 256*DL5 is the address of the first screen memory address instruction in the display list; MENU is the address of the keyboard layout display; DLMEN is the display list instruction giving the low byte of the address of the keyboard layout display; DISPLAY is the address of the machine language scroll subroutine; WHI is the high byte of WIDE; WLO is the low byte of WIDE.

655-665. Set up the display list for screen memory.

670-675. Set up the display list for the keyboard layout display window.

680-700. End the display list; load the scrolling subroutine from disk to DISPLAY; set up the parameters DISPLAY will use (WHI and WLO); and POKE the address of the display list into locations 560 and 561.

```
655 DL=256*(TOP+4):DL4=DL+4:DL5=DL+5
    :FOR I=0 TO 2:POKE DL+I,112:NEXT
     I:PIX=0:N=0
660 FOR I=DL+3 TO DL+27+36*(MODE<>5)
     STEP 3:C=SC+N*WIDE:POKE I,64+MO
    DE:POKE I+2,INT(C/256)
665 POKE I+1,C-256*PEEK(I+2):N=N+1:N
    EXT I
670 N=0:MENU=256*(TOP+5)+64:DLMEN=DL
    +32+36*(MODE<>5):POKE DLMEN-2,MO
    DE+64:POKE DLMEN,INT(MENU/256)
675 POKE DLMEN-1,MENU-256*PEEK(DLMEN
    ):FOR I=DLMEN+1 TO DLMEN+3:POKE
    I,MODE:NEXT I
680 POKE I,65:POKE I+1,0:POKE I+2,DL
    /256:OPEN #1,4,0,"D:DISPLAY.SUB"
685 DISPLAY=DL+128:TRAP 690:FOR I=0
    TO 186:GET #1,N:POKE DISPLAY+I,N
    :NEXT I:GOTO 695
690 POP
695 WHI=INT(WIDE/256):WLO=WIDE-256*W
    HI:POKE DL+112,WLO:POKE DL+113,W
    HI
700 POKE 560,0:POKE 561,DL/256:CLOSE
     #1:RETURN
```

Stop Edit Subroutine. This routine begins by executing the Display Save Subroutine at 20, restoring the ROM character set, and changing to GRAPHICS 0. Then the user decides whether to change the name of "D1:TEMPFILE.SCR" to FSAVE$, whether to quit or not, and whether to return to edit the same screen or start Fontbyter over.

705-740. Save the screen and get the user's choices.

745. Restore the old top of memory and start the program over.

750-755. Return to edit the same screen. The flag FSAVE tells whether the screen is on disk as "D1:TEMPFILE.SCR" or FSAVE$.

760. Restore the top of memory, clear the keyboard buffer, and quit.

765-780. Check to see if a file named FSAVE$ is already on the disk. If it is, unlock it and erase it. Then change the name of "D1:TEMPFILE.SCR" to FSAVE$.

```
7Ø5 POKE SC+PIX,OLD:GOSUB 4Ø:POKE 75
    6,OLDCHBAS:GRAPHICS Ø:POKE 764,255
71Ø ? "Screen is saved as D1:TEMPFIL
    E.SCR":? :? "Do you want to save
     the screen as":? FSAVE$;"? (Y o
    r N)"
715 GOSUB 785:ON N<>43 AND N<>35 GOT
    O 715:IF N=43 THEN GOSUB 765:GOT
    O 725
72Ø FSAVE=Ø
725 ? :? "Do you want to quit? (Y or
     N)":POKE 764,255
73Ø GOSUB 785:ON N<>43 AND N<>35 GOT
    O 73Ø:ON N=35 GOTO 735:ON N=43 G
    OTO 76Ø
735 ? :? "To return to edit the same
     screen,{4 SPACES}press OPTION":
    ? :? "To start FONTBYTER over, p
    ress START"
74Ø OPT=PEEK(53279):ON ((OPT=6)+(2*(
    OPT=3))) GOTO 745,75Ø:GOTO 74Ø
745 POKE 1Ø6,A:GRAPHICS Ø:GOTO 1Ø
75Ø POKE 1Ø6,TOP:GOSUB 635:FL$="D1:T
    EMPFILE.SCR":IF FSAVE=1 THEN FL$
    =FSAVE$
755 GOSUB 2Ø:GOSUB 655:GOTO 12Ø
76Ø POKE 1Ø6,A:POKE 764,255:GRAPHICS
     Ø:END
765 FSAVE=1:TRAP 77Ø:OPEN #2,4,Ø,FSA
    VE$:CLOSE #2:XIO 36,#2,Ø,Ø,FSAVE
    $:XIO 33,#2,Ø,Ø,FSAVE$:GOTO 775
77Ø CLOSE #2
775 FL$="D1:TEMPFILE.SCR,":FLL$=FSAV
    E$(4,LEN(FSAVE$)):FL$(17)=FLL$
78Ø XIO 32,#1,Ø,Ø,FL$:RETURN
```

Keycode Subroutine. Read the keyboard buffer at 764. C is the actual key combination, and N is the key pressed, regardless of the CONTROL and SHIFT keys.

```
785 C=PEEK(764):N=C-64*INT(C/64):RET
    URN
```

Test WIDE Subroutine. Check to make sure WIDE is within the legal range.

```
790 IF WIDE<40 THEN WIDE=40:RETURN
795 IF WIDE>170 AND MODE<>5 THEN WID
    E=170:RETURN
800 IF WIDE<340 THEN RETURN
805 WIDE=340:RETURN
```

Load DELETE$ and EXPAND$. Read these two machine language subroutines from disk files "D1:DELETE.SUB" and "D1:EXPAND.SUB", and store them as strings DELETE$ and EXPAND$.

```
810 TRAP 815:OPEN #1,4,0,"D:DELETE.S
    UB":FOR I=1 TO 124:GET #1,N:DELE
    TE$(I,I)=CHR$(N):NEXT I:GOTO 820
815 POP
820 CLOSE #1:WHI=INT(WIDE/256):WLO=W
    IDE-256*WHI
825 TRAP 830:OPEN #1,4,0,"D:EXPAND.S
    UB":FOR I=1 TO 124:GET #1,N:EXPA
    ND$(I,I)=CHR$(N):NEXT I:GOTO 835
830 POP
835 CLOSE #1:RETURN
```

Get Directory Subroutine. Get a directory of disk files with a particular extender and display it. Entry point 840 gets a directory of SCReen files, and entry point 850 gets a directory of character SETs.

```
840 TRAP 865:XIO 36,#1,0,0,"D:*.SCR"
845 ? :? "Currently saved screen fil
    es:":FLL$="SCR":GOTO 860
850 TRAP 865:XIO 35,#1,0,0,"D:*.SET"
855 ? :? "Currently available charac
    ter sets:":FLL$="SET"
860 FL$="D1:*.":FL$(LEN(FL$)+1)=FLL$
    :OPEN #1,6,0,FL$:FOR I=0 TO 50:I
    NPUT #1,FLL$:? FLL$:NEXT I
865 CLOSE #1:RETURN
```

Quick Scroll Subroutine. This routine eliminates the cursor and reads the joystick to determine the direction of the scroll. It does not change screen memory, only the area of screen memory being read by the display list. Pressing START ends this routine.

```
870 GOSUB 920:POKE SC+PIX,OLD:GOTO 8
    95
875 WV=2*((DI=5)+(DI=13)+(DI=9))+(DI
    =10)+(DI=6)+(DI=14):WH=2*(DI<8 A
    ND DI>4)+(DI<12 AND DI>8)
880 W=(PEEK(DL4)+256*PEEK(DL5))-SC:U
    =INT(W/WIDE):WV=WV-(U=0 AND WV=1
    )-2*((U+7+12*(MODE<>5)=LINE-2) A
    ND WV=2)
885 L=W-(U*WIDE):WH=WH-(L=0 AND WH=1
    )-2*((L+40)=WIDE AND WH=2)
890 POKE DL+114,WH:POKE DL+115,WV:C=
    USR(DISPLAY)
895 IF PEEK(53279)<>6 THEN DI=PEEK(6
    32):ON DI<>15 GOTO 875:GOTO 895
900 PIX=PEEK(DL4)+256*PEEK(DL5)+(6+6
    *(MODE<>5))*WIDE+20:OLD=PEEK(PIX
    ):PIX=PIX-SC:RETURN
```

Keycode Array Subroutine. This routine creates an array C(*n*) such that in each array element C(X), X is the keyboard code read from 764 and C(X) is the internal character code value for the key depressed. The values are read from the disk file "D1:CHARDATA.DAT".

```
905 OPEN #4,4,0,"D:CHARDATA.DAT"
910 FOR I=0 TO 255:GET #4,N:C(I)=N:N
    EXT I
915 CLOSE #4:RETURN
```

Delay Subroutine. This routine sounds the console buzzer and takes up time, so that users have time to lift their fingers from the keyboard to avoid repeating a command.

```
920 FOR I=0 TO 10:POKE 53279,4:NEXT
    I:RETURN
```

Load Keyboard Display. This routine sets up the keyboard display on the bottom four lines of the screen. If the routine is entered at line 930, it causes the currently available character group (MENSH) to be displayed.

```
925 OPEN #1,4,0,"D:MENU.DAT":FOR I=4
     TO 483:GET #1,N:POKE MENU+I,N:N
    EXT I:CLOSE #1
930 MENSH=MENU+160*INT(SHIF/64):POKE
     DLMEN,INT(MENSH/256):POKE DLMEN
    -1,MENSH-256*PEEK(DLMEN):RETURN
```

Compile Fontbyter

Fontbyter will run much more quickly if the BASIC program is compiled. Program 8 is listed here for those users who wish to make a compiled version. If the lines from Program 8 are added to or substituted for lines in Fontbyter, the program will compile using Monarch Data Systems' *ABC* Compiler.

Program 2. Machine Language Scrolling Subroutine

```
900 OPEN #1,8,0,"D1:DISPLAY.SUB"
910 FOR I=1 TO 186:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 104,173,49,2,133,206,133,2
     13
1008 DATA 173,48,2,105,3,133,205,105
1016 DATA 109,133,212,162,0,161,205,
     41
1024 DATA 191,133,207,230,205,161,20
     5,133
1032 DATA 203,160,1,177,205,133,204,
     200
1040 DATA 177,212,240,34,201,2,208,1
     6
1048 DATA 24,165,203,105,1,133,203,1
     65
1056 DATA 204,105,0,133,204,24,144,1
     4
1064 DATA 56,165,203,233,1,133,203,1
     65
1072 DATA 204,233,0,133,204,24,160,3
1080 DATA 177,212,240,42,201,2,208,1
     9
1088 DATA 24,165,203,160,0,113,212,1
     33
1096 DATA 203,200,165,204,113,212,13
     3,204
1104 DATA 24,144,19,56,165,203,160,0
1112 DATA 241,212,133,203,165,204,20
     0,241
1120 DATA 212,133,204,24,144,0,160,8
1128 DATA 165,207,201,5,240,2,160,20
1136 DATA 162,0,165,203,129,205,230,
     205
1144 DATA 165,204,129,205,132,207,24
     ,165
1152 DATA 203,160,0,113,212,133,203,
     165
1160 DATA 204,200,113,212,133,204,23
     0,205
```

```
1168 DATA 230,205,164,207,136,208,21
     9,165
1176 DATA 203,129,205,230,205,165,20
     4,129
1184 DATA 205,96
```

Program 3. Machine Language Line Insert Subroutine

```
900 OPEN #1,8,0,"D1:EXPAND.SUB"
910 FOR I=1 TO 122:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 104,166,207,169,0,165,209,
     240
1008 DATA 29,160,255,177,203,145,205
     ,136
1016 DATA 208,249,230,204,230,206,16
     4,208
1024 DATA 177,203,145,205,136,208,24
     9,198
1032 DATA 204,198,206,24,144,9,164,2
     08
1040 DATA 177,203,145,205,136,208,24
     9,202
1048 DATA 240,29,56,165,205,229,208,
     133
1056 DATA 205,165,206,229,209,133,20
     6,56
1064 DATA 165,203,229,208,133,203,16
     5,204
1072 DATA 229,209,133,204,24,144,182
     ,165
1080 DATA 209,240,27,160,255,169,0,1
     45
1088 DATA 203,136,208,251,230,206,23
     0,204
1096 DATA 164,208,145,203,136,208,25
     1,198
1104 DATA 206,198,204,24,144,11,164,
     208
1112 DATA 240,7,169,0,145,203,136,20
     8
1120 DATA 251,96
```

Program 4. Machine Language Line Delete Subroutine

```
900 OPEN #1,8,0,"D1:DELETE.SUB"
910 FOR I=1 TO 122:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 104,166,207,169,0,165,209,
     240
```

```
1008 DATA 29,160,255,177,205,145,203
     ,136
1016 DATA 208,249,230,204,230,206,16
     4,208
1024 DATA 177,205,145,203,136,208,24
     9,198
1032 DATA 204,198,206,24,144,9,164,2
     08
1040 DATA 177,205,145,203,136,208,24
     9,202
1048 DATA 240,29,24,165,205,101,208,
     133
1056 DATA 205,165,206,101,209,133,20
     6,24
1064 DATA 165,203,101,208,133,203,16
     5,204
1072 DATA 101,209,133,204,24,144,182
     ,165
1080 DATA 209,240,27,160,255,169,0,1
     45
1088 DATA 205,136,208,251,230,206,23
     0,204
1096 DATA 164,208,145,205,136,208,25
     1,198
1104 DATA 206,198,204,24,144,11,164,
     208
1112 DATA 240,7,169,0,145,205,136,20
     8
1120 DATA 251,96
```

Program 5. Data for Keyboard Display

```
900 OPEN #1,8,0,"D1:MENU.DAT"
910 FOR I=1 TO 482:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 0,0,91,0,17,0,18,0
1008 DATA 19,0,20,0,21,0,22,0
1016 DATA 23,0,24,0,25,0,16,0
1024 DATA 28,0,30,0,126,0,0,0
1032 DATA 0,0,0,0,0,0,0,0
1040 DATA 0,0,0,127,0,113,0,119
1048 DATA 0,101,0,114,0,116,0,121
1056 DATA 0,117,0,105,0,111,0,112
1064 DATA 0,13,0,29,0,0,0,0
1072 DATA 0,0,0,0,0,0,0,0
1080 DATA 0,0,0,0,0,0,97,0
1088 DATA 115,0,100,0,102,0,103,0
1096 DATA 104,0,106,0,107,0,108,0
1104 DATA 27,0,11,0,10,0,0,0
```

```
1112 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1120 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,122
1128 DATA Ø,120,Ø,99,Ø,118,Ø,98
1136 DATA Ø,110,Ø,109,Ø,12,Ø,14
1144 DATA Ø,15,Ø,Ø,Ø,Ø,Ø,Ø
1152 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1160 DATA Ø,Ø,Ø,Ø,1,Ø,2,Ø
1168 DATA 3,Ø,4,Ø,5,Ø,6,Ø
1176 DATA 7,Ø,32,Ø,8,Ø,9,Ø
1184 DATA 125,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1192 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1200 DATA Ø,Ø,Ø,Ø,Ø,49,Ø,55
1208 DATA Ø,37,Ø,50,Ø,52,Ø,57
1216 DATA Ø,53,Ø,41,Ø,47,Ø,48
1224 DATA Ø,63,Ø,124,Ø,Ø,Ø,Ø
1232 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1240 DATA Ø,Ø,Ø,Ø,Ø,Ø,33,Ø
1248 DATA 51,Ø,36,Ø,38,Ø,39,Ø
1256 DATA 40,Ø,42,Ø,43,Ø,44,Ø
1264 DATA 26,Ø,60,Ø,62,Ø,Ø,Ø
1272 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1280 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,58
1288 DATA Ø,56,Ø,35,Ø,54,Ø,34
1296 DATA Ø,46,Ø,45,Ø,59,Ø,61
1304 DATA Ø,31,Ø,Ø,Ø,Ø,Ø,Ø
1312 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1320 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1328 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1336 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1344 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1352 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1360 DATA Ø,Ø,Ø,Ø,Ø,81,Ø,87
1368 DATA Ø,69,Ø,82,Ø,84,Ø,89
1376 DATA Ø,85,Ø,73,Ø,79,Ø,80
1384 DATA Ø,92,Ø,93,Ø,Ø,Ø,Ø
1392 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1400 DATA Ø,Ø,Ø,Ø,Ø,Ø,65,Ø
1408 DATA 83,Ø,68,Ø,70,Ø,71,Ø
1416 DATA 72,Ø,74,Ø,75,Ø,76,Ø
1424 DATA 123,Ø,94,Ø,95,Ø,Ø,Ø
1432 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1440 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,90
1448 DATA Ø,88,Ø,67,Ø,86,Ø,66
1456 DATA Ø,78,Ø,77,Ø,64,Ø,96
1464 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1472 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1480 DATA Ø,Ø
```

Program 6. Data for Keycode Array

```
900 OPEN #1,8,0,"D1:CHARDATA.DAT"
910 FOR I=1 TO 256:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 108,106,27,0,0,107,11,10
1008 DATA 111,0,112,117,0,105,13,29
1016 DATA 118,0,99,0,0,98,120,122
1024 DATA 20,0,19,22,91,21,18,17
1032 DATA 12,0,14,110,0,109,15,0
1040 DATA 114,0,101,121,127,116,119,
     113
1048 DATA 25,0,16,23,126,24,28,30
1056 DATA 102,104,100,0,0,103,115,97
1064 DATA 44,42,26,0,0,43,60,62
1072 DATA 47,0,48,53,0,41,63,124
1080 DATA 54,0,35,0,0,34,56,58
1088 DATA 4,0,3,6,0,5,2,1
1096 DATA 59,0,61,46,0,45,31,0
1104 DATA 50,0,37,57,0,52,55,49
1112 DATA 8,0,9,7,0,32,125,0
1120 DATA 38,40,36,0,0,39,51,33
1128 DATA 76,74,123,0,0,75,94,95
1136 DATA 79,0,80,85,0,73,92,93
1144 DATA 86,0,67,0,0,66,88,90
1152 DATA 0,0,0,0,0,0,0,0
1160 DATA 64,0,96,78,0,77,0,0
1168 DATA 82,0,69,89,0,84,87,81
1176 DATA 0,0,0,0,0,0,0,0
1184 DATA 70,72,68,0,0,71,83,65
1192 DATA 0,0,0,0,0,0,0,0
1200 DATA 0,0,0,0,0,0,0,0
1208 DATA 0,0,0,0,0,0,0,0
1216 DATA 0,0,0,0,0,0,0,0
1224 DATA 0,0,0,0,0,0,0,0
1232 DATA 0,0,0,0,0,0,0,0
1240 DATA 0,0,0,0,0,0,0,0
1248 DATA 0,0,0,0,0,0,0,0
```

Program 7. Clear Subroutine

```
900 OPEN #1,8,0,"D1:CLEAR.SUB"
910 FOR I=1 TO 33:READ N:PUT #1,N:NE
    XT I:CLOSE #1:? I:END
1000 DATA 104,104,104,133,208,104,10
     4,101
1008 DATA 208,133,209,169,0,133,207,
     160
1016 DATA 255,145,207,136,208,251,14
     5,207
```

```
1Ø24 DATA 23Ø,2Ø8,165,2Ø8,197,2Ø9,2Ø
     8,235
1Ø32 DATA 96
```

Program 8. Altered Lines for Compiler Version

```
215 GOSUB 935:RETURN
275 ON (4*T)+DI GOSUB 28Ø,285,29Ø,29
    5,3ØØ,3Ø5,31Ø,32Ø:GOSUB 935:GOTO
     265
78Ø POKE ICCOM+X,32:I=ADR(FL$):POKE
    ICBADR+X+1,INT(I/256)
781 POKE ICBADR+X,I-256*PEEK(ICBADR+
    X+1):POKE ICBLEN+X,LEN(FL$):POKE
     ICBLEN+X+1,Ø
782 I=USR(ADR("hhh▣LV▣"),X):RETURN
92Ø FOR I=Ø TO 12Ø:POKE 53279,4:NEXT
     I:RETURN
935 FOR I=Ø TO 6Ø:NEXT I:RETURN
```

5

Screenbyter

Carl Zahrt and Orson Scott Card

Here is an artist's utility that lets you create screen displays in any of the regular pixel graphics modes—and GRAPHICS "6.5" (XL Mode 14) and "7.5" (XL Mode 15) as well. It's simple enough for a child to use. It gives you complete control over color, mode, and display size. And a special Fill Mode lets you draw long lines or fill large areas with color in moments. "Screenbyter" requires 40K memory and a disk drive.

For color graphics, the Atari home computers are superb. Creating screen displays from BASIC, page flipping, scrolling, redefining characters, continuous memory, and changing from mode to mode to get exactly the effect you want—once you've worked with graphics on the Atari, most other home computers feel a bit confining.

But that doesn't mean using Atari graphics is *easy*, especially if you want large displays which extend far beyond the edges of the TV screen, or detailed drawings that would take hundreds of PLOT and DRAWTO statements to create from BASIC. Such things take painstaking work on graph paper and many POKEs into screen memory—or a good chunk of your paycheck for software to do it for you.

"Screenbyter" takes the pain out of creating beautiful graphics displays.

- You can work in any of the non-GTIA pixel modes.
- You have access to GRAPHICS "6.5" and "7.5" pixel modes that cannot be used with a simple GRAPHICS statement.
- You can type RUN and start drawing with the joystick—no programming experience is needed.
- You can fill in large areas quickly and easily.
- Since the main action of the program is in machine language, it moves very quickly, but a Slow Mode is provided so you can do detail work, pixel by pixel.
- You can change screen colors with the joystick.
- You aren't always limited by the size of the screen. In GRAPHICS 3 you can create scrolling displays many times

larger than the TV screen, and all the modes except 7.5 and 8 allow some scrolling.

• When you save a display to disk, all the parameters—mode, size, and colors—are saved with the screen data, so that you can load them directly into your own programs.

Using Screenbyter

Setup. Screenbyter begins by displaying a directory of all files on the disk with the extender ".PIX". This extender is automatically added to all files created by Screenbyter. If no directory appears, there are no previously saved files on the disk.

"What file should hold your finished screen? (Eight characters)." Respond to this prompt by giving the filename you want your new display to have, when you save it at the end of the editing session. Screenbyter automatically removes everything before a colon or after a period and replaces the characters with "D1:" and ".PIX", so that you need to enter only the eight-letter filename. If you use illegal characters, Screenbyter will ask you to try again; if you use more than eight characters, only the first eight characters will be used.

If the name you enter is the name of a file already on disk, Screenbyter will remind you of that. To change the name, press RETURN. Or, if you want your new display to overwrite the old file, press any other key to go on.

"Would you like to edit a screen you have already saved? (Y or N)". If you answer Y, Screenbyter asks you for the name of the saved file. If the file is not on disk in the form "D1:*filename*.PIX", Screenbyter will tell you and ask you to insert the correct disk or, if you wish, ask you again if you want to edit a previously saved screen.

Once the file is found, Screenbyter reads the first four bytes of the file to get the mode number, the number of bytes per line, and the number of lines in the display as it was saved. Press RETURN if you want to change these parameters. Press any other key to leave them the same.

Changing the parameters can have interesting effects. Remember that four-color modes all read the bytes the same way; if you want to draw your displays in GRAPHICS 3 (ANTIC 8) and then display them in a higher four-color mode, you can. Changing the length of a file either chops off the bottom or adds blank lines at the bottom of the display. Changing the line width, however, will usually result in garbage, since the vertical relationships will all be changed. The option is included, however, because sometimes garbage can be fun, and who are we to forbid you to change the line width just because we can't think of a reason for doing it?

If you are not editing a previously saved display, or if you are changing the parameters, you get the following series of prompts:

"What Antic mode will you work in?" This prompt is followed by a table, which lists the eight ANTIC pixel modes and their graphics mode equivalent. ANTIC 8, for instance, is GRAPHICS 3; ANTIC F (15) is GRAPHICS 8. Two ANTIC modes, C (12) and E (14), have no graphics equivalent—they are the famous "GRAPHICS 6.5" and "GRAPHICS 7.5." (See Table 1.) Enter the ANTIC mode number: 8, 9, A (10), B (11), C (12), D (13), E (14), or F (15).

"How wide a line? (Minimum *nn* byte, maximum *nn* bytes)." Depending on the mode you choose, Screenbyter will give you minimum and maximum number of bytes per line. Remember that in the four-color modes, each byte is four pixels, while in the two-color modes, each byte is eight pixels. The minimum is based on the minimum number of bytes required to fill the screen. The maximum is based on the widest possible line that will allow the display to fit within 4K. If you enter numbers outside the legal range, Screenbyter will select the minimum or maximum, as appropriate.

With ANTIC *E* and *F*, the minimum and maximum are the same—you have no option, so any number you enter will result in the same number of bytes per line. This is because these two modes will not scroll—they both require more than 4K. Scrolling a screen that crosses a 4K boundary requires elaborate arrangements of screen memory that were beyond the scope of this program. Displays created in *E* and *F* will take up 65 sectors on disk; all other displays will take up 33 sectors or fewer.

"How many lines do you want to edit? (Minimum *nn*, maximum *nn*)". The minimum and maximum depend on the mode and the number of bytes per line already selected. Again, if you choose parameters outside the legal range, Screenbyter will select the minimum or maximum. And if you choose the maximum number of bytes per line, only the minimum number of lines per screen will be possible.

When all selections have been made, you are given one last chance to change your mind. All the parameters you chose are displayed on the screen. If these are correct, press START, and the program will go on. If you want to make changes, press OPTION and the program will start over.

Waiting. What's going on while you wait? Screenbyter configures the memory to reserve 10K (40 pages) at the top of memory to hold screen memory (up to 8K), the display list, and the machine language routine that actually puts your drawing on the screen. Screen memory is cleared and the machine language routines are loaded. If you chose to edit a previously saved screen, it is loaded into memory now. All this takes about six seconds. The rest of the time is spent writing the display list. The higher the ANTIC mode, the longer it takes to write the display list—ANTIC F requires about 200 POKEs in BASIC, plus the calculations to find out what numbers to POKE, and it can take as long as 20 seconds.

When Screenbyter is ready for you to edit, there will be a cursor in the upper-left-hand corner.

Moving the cursor. The joystick controls the cursor.

Drawing a line. Hold down the joystick button to draw; let it up to move the cursor without drawing.

Selecting a color. Press 1 or SHIFT-CAPS/LOWR to select Color 1. Press 2 or CONTROL-CAPS/LOWR for Color 2. Press 3 or SHIFT-CONTROL-CAPS/LOWR for Color 3. Press 0 or CAPS/LOWR to select the background color. Drawing in the background color has the effect of erasing.

Color Mode. To change the actual colors that are displayed by Colors 1, 2, or 3, or the background color, press START. You will hear a buzz, and the cursor will no longer respond to the joystick. Instead, moving the joystick will change the colors displayed on the screen. Moving the joystick up or right will change the color from darker to brighter, then jump to the darkest value of the next color. Moving the joystick down or left will change the color from brighter to darker, then jump to the brightest value of the next color.

To change the background color, move the joystick forward or back; to change Color 3, move the joystick left or right. To change Color 2, move the joystick forward or back with the button pressed; to change Color 1, move the joystick left or right with the button pressed.

To return to Cursor Mode, press START again. No other commands will work during Color Mode.

Slow Mode. Press the space bar to enter Slow Mode. A delay loop in the program makes the cursor move much more slowly around the screen, with a click between moves. This mode allows you to create details. To return to Fast Mode, press the space bar again.

Fill Mode. Press the inverse key (Atari logo key) to enter Fill Mode. A low hum will come from the television. In this mode, when you press the joystick button, Screenbyter draws a dot of the selected color at the current cursor location, as usual, but it also searches to the right along the same line. If it finds another dot of the same color before it reaches the end of the line, it will fill in all the area between that dot and the current cursor position with dots of the same color. If no dot of the same color is found, no fill operation is performed.

This allows you to fill large or small areas of the screen with a single color. Simply draw the right-hand edge of the figure first; then enter Fill Mode and draw the left-hand border. It takes some practice to use this function without accidentally erasing parts of your screen, but you will probably find that this is the most useful feature of Screenbyter.

To exit Fill Mode, press the inverse key again. The hum will continue as long as you are in Fill Mode, and will stop only when you leave.

Insert a line. Press SHIFT-INSERT to insert a line at the current cursor position. The bottom line of the display will be pushed down and lost.

Delete a line. Press SHIFT-DELETE to delete the current cursor line. A blank line will be added at the bottom of the display.

Clear the screen. Press CONTROL-SHIFT-CLEAR to completely erase the screen. If you haven't already saved the display, it will be lost.

Saving the screen. Press SELECT to save the screen without ending the editing session. The current screen display will be saved as "D1:TEMPFILE.PIX". You can save as often as you like; Screenbyter will simply overwrite any existing TEMPFILE.PIX file.

Ending the editing session. Press OPTION to save the screen and end the editing session. (To exit without saving, press RESET.) The display will be saved as "D1:TEMPFILE.SCR." Then the regular GRAPHICS 0 screen will return, and you will be given several prompts:

"Do you want to save the screen as D1:*filename*.PIX? (Y or N)". If you answer *N*, the saved display will be left as TEMPFILE.PIX. If you answer *Y*, Screenbyter will erase any existing file that has the same filename. Then Screenbyter will rename TEMPFILE.PIX with the filename you chose.

"Do you want to quit? (Y or N)". If you answer *Y*, Screenbyter will restore the old top of memory and exit to BASIC. If you answer *N*, you will get another prompt. To return to edit the screen you just left, press OPTION. That display will be reloaded into memory, the display list will be rewritten, and you can start over. To edit an entirely new screen, or to change the name of the save file, press START. In effect, Screenbyter will then start over.

What's Going On inside Screenbyter?

Like everything else in a computer, your display exists as a series of numbers stored in binary form in memory locations in the computer. The ANTIC chip scans screen memory as it is instructed to do by the display list. But it doesn't read the numbers as numbers. Instead, it reads them as patterns of "on" and "off" bits.

Four-color modes. In the four-color modes, each byte is read as code for four pixels. The eight-bit binary number is treated as four bit-pairs:

00 00 00 00

Each bit-pair provides the code for one pixel, or rectangle of color on the screen. In GRAPHICS 3, each pixel is the size of a character in GRAPHICS 0. In GRAPHICS 7.5, each pixel is one scan line high and two color clocks wide, which gives very good resolution. But all four-color modes read the bit-pairs the same way.

00 means to display the background color (the color code stored at location 712).

01 means to display Color 1 (the color code stored at location 708).

10 means to display Color 2 (the color code stored at location 709).

11 means to display Color 3 (the color code stored at location 710).

This means that the number 216 (binary 11011000) is treated as four pixel-color instructions: The first pixel is Color 3, the second pixel is Color 1, the third pixel is Color 2, and the last pixel is the background color.

Two-color modes. The two-color modes treat each bit as a separate pixel instruction, so that each byte controls eight pixels. An "on" bit, or 1, is read as a Color 1 instruction, while an "off" bit, or 0, is read as a background color instruction. In a two-color mode, the number 216 would be treated as eight pixel-color instructions: Two "on" pixels, one "off" pixel, two more "on" pixels, and three "off" pixels. (See Table 1 for a listing of all the modes.)

Table 1. Pixel Modes

ANTIC mode	8	9	A (10)	B (11)	C (12)	D (13)	E (14)	F (15)
Graphics mode	3	4	5	6	—	7	—	8
Colors	4	2	5	2	2	4	4	2
Resolution	24 x	48 x	48 x	96 x	192 x	96 x	192 x	192 x
	40	80	80	160	160	160	160	320
Memory, bytes	240	480	960	1920	3840	3840	7680	7680
(sectors)	(3)	(5)	(9)	(17)	(33)	(33)	(65)	(65)
Lines/screen	24	48	48	96	192	96	192	192
Bytes/line	10	10	20	20	20	40	40	40
Bits/pixel	2	1	2	1	1	2	2	1
(Pixels/byte)	(4)	(8)	(4)	(8)	(8)	(4)	(8)	(4)
Scan lines/pixel	8	4	4	2	1	2	1	1
Color clocks/								
pixel	4	2	2	1	1	1	1	1/2

Note: ANTIC C (12) and E (14), the two "hidden" pixel modes, provide the same resolution. All the other pixel modes attempt to create as square a pixel as the TV screen allows—the same number of color clocks wide as scan lines high. *C* and *E*, however, are twice as wide as they are high, making each pixel very short and wide. They come very near the resolution of ANTIC F (GRAPHICS 8). The advantages are that, compared to *F*, *C* uses half the memory and *E* allows four colors.

Moving around the screen. Moving the cursor around the screen, then, isn't simply a matter of moving from one byte to the next in screen memory. Screenbyter also has to move from bit to bit or from bit-pair to bit-pair within the bytes. This *can* be done in BASIC by adding or subtracting values, but it is very slow. Machine language, however, has powerful commands that make it easy to move from bit to bit. DRAWTO and PLOT commands do these manipulations for you, but since Screenbyter is circumventing the BASIC graphics commands entirely, the main drawing operations are executed in machine language.

To understand what Screenbyter is doing, you need to understand a few machine language commands: EOR, ORA, and AND. The two OR instructions and the AND instruction are not the same as the AND and OR you use in Atari BASIC. In machine language, these are operations on the bits of an eight-bit number, and are often called "bitwise" AND and OR to help keep the difference in mind.

All three operations compare two numbers, one stored in the accumulator and another somewhere else in memory. The operation results in a third number, which is stored in the accumulator in place of the number that was already there.

AND, referred to as "bitwise AND," compares the two numbers, bit by bit. Any bit that is on in both numbers *stays* on in the resulting number. All other bits are turned off. In other words, only bits that are on in the first number *and* in the second number remain on in the result.

```
                10010110
       AND      11110000
results in      10010000
```

ORA, referred to as "bitwise OR," compares the two numbers, but in this case any bit that is on in *either* number stays on in the result:

```
                10010110
       ORA      11110000
results in      11110110
```

EOR, referred to as "exclusive OR," compares the two numbers, and any bit that is on in one and *only* one number is left on in the result. Any bit that is on in both numbers or off in both numbers is off in the result:

```
                10010110
       EOR      11110000
results in      01100110
```

How do these actually work, in practice?

Screenbyter maintains several *masks.* The Color Mask is in page six, at memory location 1692 ($069C). This byte is set from BASIC whenever the color is changed, and it is set so that every bit or bit-pair represents a pixel of the selected color. If the background color is selected, the Color Mask is 00000000. If Color 1 is selected, the Color Mask is 01010101. For Color 2, the Color Mask is 10101010, and for Color 3 it is 11111111. With two-color modes, the Color Mask is either 00000000 or 11111111.

The Cursor Mask is kept at location 1696 ($06A0). It is set to represent the current cursor pixel within the cursor byte. The bits in the current pixel are on; all others are off. In four-color modes, if the cursor is in the leftmost pixel of the cursor byte, the Cursor Mask will be set to 11000000; if it is in the rightmost pixel, the mask will be set to 00000011. The two middle pixels are 00110000 and 00001100. In two-color modes, a single "on" bit represents the cursor position.

Whenever you move the cursor left or right or diagonally, the Cursor Mask is shifted left or right, so that at any given moment the Cursor Mask will mark which bit or bit-pair Screenbyter should change.

If you are drawing, Screenbyter first picks up the value of the current cursor byte and stores it at 1690 ($069A). Then it picks up the Cursor Mask and EORs it with 11111111 (decimal 255). This reverses the Cursor Mask—any bit that was on is now off, and any bit that was off is now on.

Let's see that in action in a four-color mode, in which the background is black, Color 1 is red, Color 2 is green, and Color 3 is blue. The bit-pairs will be separated in these examples, to make it easier to keep track of the pixels.

```
Cursor Mask   00 11 00 00
        EOR   11 11 11 11
 results in   11 00 11 11   (Reverse Cursor Mask)
```

Screenbyter then ANDs the Reverse Cursor Mask with the number at 1690 ($069A), which in effect makes a hole in the cursor position:

```
Reverse Cursor Mask   11 00 11 11
                AND   01 01 01 11    red    red    red    blue
         results in   01 00 01 11    red    —      red    blue
```

The two bits in the cursor position will *always* be turned off.

Now Screenbyter must prepare the pixel code to go in that hole. Screenbyter picks up the Cursor Mask and ANDs it with the Color Mask. Since all the bits in the Cursor Mask are off except the two bits of the current pixel, the resulting number will have only the bits that represent the current color, and only in the pixel position:

```
   Cursor Mask   00 11 00 00
AND Color Mask   10 10 10 10    green   green   green   green
    results in   00 10 00 00    —       green   —       —
```

Now we are ready to put the correct pixel code into the hole in the current cursor byte. To do this, we bitwise OR the current pixel (the one with the cursor byte with a hole in it) we just got from the operation before. Remember that with ORA, any byte that is on in either or both of the two numbers is on in the result:

correct pixel	00 10 00 00	—	green	—	—
ORA current byte with hole	01 00 01 11	red	—	red	blue
results in	01 10 01 11	red	green	red	blue

The result is then stored in 1690 ($069A), and later in the program it is put into screen memory.

If you are not drawing, but just moving the cursor, the operation is a little different, but AND, EOR, and ORA perform the same functions.

Machine language is so fast that all this seems to happen instantaneously. In fact, the only reason the cursor doesn't fly around the screen out of control is because Screenbyter keeps leaving the machine language routine, returning to BASIC to check the keyboard for other commands. Even so, the cursor moves so quickly that it has to be slowed down in order to allow you to draw details.

Use of page six. The machine language routine at SCROLL uses a field in page six to hold some important variables. The memory locations in page six are explained in Table 2.

Screenbyter Displays in Your Own Programs

Here are two routines you can add to your own programs, which will allow you to load the displays you created with Screenbyter. The first routine, Load and Display List, works with any Screenbyter file. However, it sets up a custom display list with individual LMS instructions, suitable for scrolling. This makes the set-up time rather long. So a Simple Load Routine is also included. It will work with any display file that was created using the *minimum* line width and number of lines per screen, except screens created in ANTIC C and E (GRAPHICS "6.5" and "7.5"). You cannot use it if you intend to scroll horizontally. However, you *can* use it if you intend to scroll vertically or flip pages, and your display was created with the minimum line width.

Both routines will configure memory to protect the screen display, read the display parameters from whatever display file you choose, and load the file into memory. It uses a load routine very similar to the one used by Fontbyter, so we won't explain them again here.

Notice that in loading displays created in ANTIC E (14) and F (15) (GRAPHICS 7.5 and 8), the screen display must cross a 4K

Table 2. Page 6 Memory Locations

Dec	Hex	
1670	0686	WIDE – 1. Used to check for the end of the logical line.
1671	0687	Used in fill routine to keep track of right border of fill.
1672	0688	Cursor location: current byte on logical line.
1673	0689	Used by the fill routine to hold the pattern of the rightmost byte of the fill line.
1674-1675	068A-068B	LINE – 1. Used to check for last line of display.
1676-1677	068C-068D	Cursor location: current logical line number.
1678	068E	Bytes per screen line – 1. Used by the scrolling routine to check for the end of the screen line.
1679	068F	Cursor location: current byte on screen line.
1680	0690	Lines per screen – 1. Used by the scrolling routine to check for the bottom of the screen display.
1681	0691	Cursor location: current screen line number.
1682	0692	Used by the fill routine to hold the pattern of the leftmost byte of the fill line.
1683	0693	A temporary holding location.
1684	0694	Used by the fill routine to hold the real value of the byte currently being tested.
1685	0695	A temporary holding location.
1686-1687	0696-0697	The current screen starting address (the address of the upper-left-hand corner of the screen).
1688-1689	0698-0699	Cursor location: the address of the current cursor byte in screen memory.
1690	069A	The real contents of the current cursor byte.
1691	069B	The reverse (cursor display) contents of the current cursor byte.
1692	069C	Color Mask.
1693	069D	The number of bits per pixel (1 or 2).
1694	069E	Scroll flag (0 = do not scroll).
1695	069F	Fill flag (0 = do not fill).
1696	06A0	Cursor Mask.
1697	06A1	Joystick value.
1698	06A2	Total number of lines per screen. Used in the scroll routine to change the correct number of LMS instructions in the display list.
1699	06A3	WIDE. Used in the scroll routine to increment the LMS addresses in the display list.
1700	06A4	Fill Test Mask. Used in the fill routine to isolate and test each pixel until a pixel of the selected color is found.
1701	06A5	Starting Fill Test Mask. Either 192 (four-color mode) or 128 (two-color mode).
1702-1704	06A6-06A8	Machine language jump vector: JMP followed by the address of the fill subroutine held in the string FILL$.

boundary line. The ANTIC chip gets fussy at this point, and ignores anything after a 4K boundary line until the beginning of the line pointed to by the next LMS instruction. Therefore, screen memory must be arranged so that the 4K boundary line comes right at the end of a line; the display list routine will have set the value of SC, the start of screen memory, so that the 4K boundary line will fall right at the end of a line.

If you have an XL model (600XL, 800XL, 1200XL, 1400XL, or 1450XLD), ANTIC C and E can be accessed from BASIC using the statements GRAPHICS 14 and GRAPHICS 15, respectively.

Program 1. Load and Display List Routine

```
5 CLR :DIM PPB(7),BPL(7),MXW(7),LPS(
  7),FL$(20):FL$="D1:SHIP.PIX":GOSUB
   4000
4000 FOR I=0 TO 7:READ W,N,C,T:PPB(I
     )=W:BPL(I)=N:MXW(I)=C:LPS(I)=T:
     NEXT I
4005 A=PEEK(106):TOP=A-36:SP=TOP+4:S
     C=SP*256:DL=256*TOP:POKE 106,TO
     P:GRAPHICS 0:PRINT "{CLEAR}"
4010 X=16:ICCOM=834:ICBADR=836:ICBLE
     N=840:SCON=PEEK(559):K4=4096
4015 OPEN #1,4,0,FL$:GET #1,M:M8=M-8
     :GET #1,WIDE:GET #1,LLO:GET #1,
     LHI:LINE=LLO+256*LHI:SZ=WIDE*LI
     NE
4020 FOR I=708 TO 711:GET #1,N:POKE
     I,N:NEXT I:POKE I,N
4025 SC=SC+((LINE*WIDE)>K4)*(K4-INT(
     K4/WIDE)*WIDE):SH=INT(SC/256):S
     L=SC-256*SH
4030 FOR I=0 TO 2:POKE DL+I,112:NEXT
      I:N=0
4035 FOR I=DL+3 TO DL+3*LPS(M8) STEP
      3:C=SC+N*WIDE:POKE I,64+M:T=IN
     T(C/256)
4040 POKE I+2,T:POKE I+1,C-256*T:N=N
     +1:NEXT I
4045 POKE I,65:POKE I+1,0:POKE I+2,D
     L/256
4050 POKE 560,0:POKE 561,DL/256
4055 POKE ICBADR+X+1,SH:POKE ICBADR+
     X,SL:POKE ICBLEN+X+1,1+INT(SZ/2
     56):POKE ICBLEN+X,0
4060 POKE ICCOM+X,7:I=USR(ADR("hhh▣L
     V▣"),X):CLOSE #1:RETURN
```

```
4Ø65 DATA 2,1Ø,17Ø,24,1,1Ø,85,48,2,2
     Ø,85,48,1,2Ø,42,96
4Ø7Ø DATA 1,2Ø,21,192,2,4Ø,42,96,2,4
     Ø,4Ø,192,1,4Ø,4Ø,192
```

Program 2. Simple Load Routine

```
5 CLR :DIM GM(15),FL$(2Ø):FL$="D1:G8
  .PIX":GOSUB 4ØØØ
6 FOR I=Ø TO 3ØØØØ:NEXT I
4ØØØ FOR I=Ø TO 15:READ N:GM(I)=N:NE
     XT I
4ØØ5 A=PEEK(1Ø6):TOP=A-36:SP=TOP+4:S
     C=SP*256:DL=256*TOP:POKE 1Ø6,TO
     P:GRAPHICS Ø:PRINT "{CLEAR}"
4Ø1Ø X=16:ICCOM=834:ICBADR=836:ICBLE
     N=84Ø:SCON=PEEK(559):K4=4Ø96
4Ø15 OPEN #1,4,Ø,FL$:GET #1,M:GET #1
     ,WIDE:GET #1,LLO:GET #1,LHI:LIN
     E=LLO+256*LHI:SZ=WIDE*LINE
4Ø2Ø FOR I=7Ø8 TO 711:GET #1,N:POKE
     I,N:NEXT I:POKE I,N
4Ø25 SC=SC+((LINE*WIDE)>K4)*(K4-INT(
     K4/WIDE)*WIDE):SH=INT(SC/256):S
     L=SC-256*SH
4Ø3Ø GRAPHICS GM(M)+16:IF GM(M)=Ø TH
     EN ? "INVALID MODE":RETURN
4Ø35 DL=PEEK(56Ø)+256*PEEK(561):DL4=
     DL+4:DL5=DL+5:POKE DL4,SL:POKE
     DL5,SH
4Ø55 POKE ICBADR+X+1,SH:POKE ICBADR+
     X,SL:POKE ICBLEN+X+1,1+INT(SZ/2
     56):POKE ICBLEN+X,Ø
4Ø6Ø POKE ICCOM+X,7:I=USR(ADR("hhh▣L
     V▣"),X):CLOSE #1:RETURN
4Ø65 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,3,4,5,6,Ø,
     7,Ø,Ø
```

Screenbyter

After the main listing of the BASIC program, you will find several programs to create disk files containing the machine language routines used in Screenbyter. If you prefer, you can easily add these DATA statements to your program and read them that way, or—as I prefer to do—load them into string constants and use them that way, without so many disk accesses. However, typing in strings that have lots of inverse and control characters in them can be tedious and often leads to typing errors, so these DATA statements are necessary in the published version of the program.

If you are also using Fontbyter, you might notice that Screenbyter follows the same structure. That's because Fontbyter was used as the starting point, and changed wherever Screenbyter's needs were different. However, the line insert, line delete, and clear screen machine language routines are *not* identical, so don't try to use the similar Fontbyter routines for Screenbyter—you will hopelessly confuse your Atari if you do, and confused Ataris have a way of locking up or otherwise expressing their frustration.

(And an acknowledgment: Our thanks to Steve and Tammy Rector at the Computer Shop in South Bend, Indiana, for their help in keeping this project from getting lost between Taiwan and Greensboro.)

Program 3. Screenbyter

```
5 DIM FSAVE$(20),FLOAD$(20),FL$(40),
  FLL$(20),DELETE$(118),EXPAND$(102)
  ,N$(13),FILL$(230),CLEAR$(26)
10 DIM PPB(7),BPL(7),MXW(7),LPS(7),C
   OL(11),CL(3)
15 A=PEEK(106):TOP=A-40:SP=TOP+8:SC=
   SP*256:DL=256*TOP:SCROLL=DL+600:P
   OKE 106,TOP
20 X=16:ICCOM=834:ICBADR=836:ICBLEN=
   840:GRAPHICS 0:SCON=PEEK(559):F=1
   670:K4=4096:N$="No equivalent"
25 C=707:FOR I=0 TO 7:IF I/2=INT(I/2
   ) THEN C=C+1:IF C=711 THEN C=712
30 COL(I)=C:NEXT I:CL(0)=0:CL(1)=85:
   CL(2)=170:CL(3)=255:FMS=ADR("hhh*
   LVE")
35 RESTORE 770:FOR I=0 TO 7:READ W,N
   ,C,T:PPB(I)=W:BPL(I)=N:MXW(I)=C:L
   PS(I)=T:NEXT I:POKE 16,112:GOTO 3
   15
40 OPEN #1,4,0,FL$:GET #1,MD:GET #1,
   WD:GET #1,LLO:GET #1,LHI:LN=LLO+2
   56*LHI:SZ=WD*LN
45 FOR I=0 TO 6 STEP 2:GET #1,N:POKE
    COL(I),N:NEXT I
50 POKE ICBADR+X+1,SH:POKE ICBADR+X,
   SL:POKE ICBLEN+X+1,1+INT(SZ/256):
   POKE ICBLEN+X,0
55 POKE ICCOM+X,7:I=USR(FMS,X):CLOSE
    #1:RETURN
60 OPEN #1,8,0,"D1:TEMPFILE.PIX":PUT
    #1,M:PUT #1,WIDE:PUT #1,LLO:PUT
   #1,LHI
```

```
65 FOR I=Ø TO 6 STEP 2:PUT #1,PEEK(C
   OL(I)):NEXT I:POKE PEEK(1688)+256
   *PEEK(1689),PEEK(169Ø)
7Ø POKE ICBADR+X+1,SH:POKE ICBADR+X,
   SL:POKE ICBLEN+X+1,1+INT((LINE*WI
   DE)/256):POKE ICBLEN+X,Ø
75 POKE ICCOM+X,11:I=USR(FMS,X):CLOS
   E #1:RETURN
8Ø IF ((LINE*WIDE-PIX)<WIDE) THEN RE
   TURN
85 C=USR(ADR(DELETE$)):POKE 169Ø,PEE
   K(PEEK(1688)+256*PEEK(1689)):POKE
    53279,4:ON SPEED GOSUB 74Ø:RETUR
   N
9Ø IF ((LINE*WIDE-PIX)<WIDE) THEN RE
   TURN
95 T=SC+WIDE*LINE-WIDE-1:C=INT(T/256
   ):T=T-256*C:POKE 2Ø5,T:POKE 2Ø6,C
1ØØ POKE (PEEK(1688)+256*PEEK(1689))
    ,PEEK(169Ø)
1Ø5 C=USR(ADR(EXPAND$)):POKE 169Ø,Ø:
    POKE 53279,4:ON SPEED GOSUB 74Ø:
    RETURN
11Ø POKE 169Ø,PEEK(SC):POKE 1691,121
    :POKE 559,SCON:OPT=8
115 OPT=PEEK(53279):IF OPT=6 THEN GO
    SUB 18Ø:GOTO 115
12Ø N=PEEK(632):C=USR(SCROLL,N):IF N
    <15 THEN POKE 77,Ø:IF SPEED THEN
     GOSUB 74Ø:POKE 53279,4
125 IF PEEK(753)=3 THEN GOSUB 14Ø:GO
    TO 115
13Ø ON OPT=3 GOTO 55Ø:IF OPT=5 THEN
    GOSUB 6Ø:GOTO 115
135 GOTO 115
14Ø GOSUB 635:ON (C=116)+2*(C=119)+3
    *(C=246) GOTO 8Ø,9Ø,17Ø
145 IF N=6Ø THEN C=C-59:SHIF=INT(C/6
    4):GOSUB 725
15Ø IF C=31 OR C=3Ø OR C=26 OR C=5Ø
    THEN GOSUB 72Ø
155 IF N=33 THEN SPEED=1*(SPEED=Ø):G
    OSUB 715
16Ø IF N=39 THEN VERS=255*(VERS=Ø):P
    OKE 1695,VERS:GOSUB 735
165 RETURN
17Ø C=USR(ADR(CLEAR$),SP):POKE 169Ø,
    Ø:POKE 1691,PEEK(1696):RETURN
175 GOSUB 715:RETURN
18Ø GOSUB 715
```

```
185 DI=PEEK(632):T=PEEK(644):DI=DI+5
    *(DI=7):DI=DI-11:OPT=PEEK(53279)
    :IF OPT=6 THEN 175
190 IF DI<Ø OR DI>3 THEN 185
195 DI=4*T+DI:IF DI/2=INT(DI/2) THEN
     POKE COL(DI),PEEK(COL(DI))-2+25
    6*(PEEK(COL(DI))<2):GOTO 185
200 POKE COL(DI),PEEK(COL(DI))+2-256
    *(PEEK(COL(DI))>253):GOTO 185
205 FLL$=FL$:FOR I=1 TO LEN(FL$):N=A
    SC(FL$(I,I)):ON N=58 GOSUB 245:N
    EXT I:FL$=FLL$
210 FLL$=FL$:FOR I=1 TO LEN(FL$):N=A
    SC(FL$(I,I)):ON N=46 GOSUB 25Ø:N
    EXT I:FL$=FLL$
215 IF LEN(FL$)>8 THEN FL$=FL$(1,8)
220 IF LEN(FL$)<1 THEN 265
225 N=ASC(FL$(1,1)):IF N>9Ø OR N<65
    THEN 26Ø
23Ø IF LEN(FL$)<2 THEN GOTO 24Ø
235 FOR I=2 TO LEN(FL$):N=ASC(FL$(I,
    I)):ON (N>9Ø OR N<65) AND (N>57
    OR N<48) GOTO 255:NEXT I
24Ø FLL$="D1:":FLL$(4)=FL$:N=Ø:RETUR
    N
245 FLL$=FL$(I+1,LEN(FL$)):RETURN
25Ø FLL$=FL$(1,I-1):RETURN
255 POP :? "{CLEAR}":? "Illegal char
    acters in ";FL$:GOTO 265
26Ø ? "{CLEAR}":? FL$;" must start w
    ith a capital":? "letter.":GOTO
    265
265 ? "Let's try that name again.":N
    =1:RETURN
27Ø TRAP 275:OPEN #1,4,Ø,FL$:N=Ø:CLO
    SE #1:RETURN
275 ? :? FL$;" isn't on disk in":? "
    drive 1":? "Insert disk with ";F
    L$;"and":? "press RETURN.":CLOSE
     #1
28Ø ? "Or to try another file name,
    press anyother key."
285 ON PEEK(753)<>3 GOTO 285:GOSUB 6
    35:ON N=12 GOTO 27Ø:N=1:RETURN
29Ø TRAP 31Ø:OPEN #1,4,Ø,FL$:? FL$;"
     is already on disk.":? "Unless
    you change the name, the old"
```

```
295 ? "file will be lost.  To change
     the namepress RETURN":? "Or pre
    ss any other key to continue.":C
    LOSE #1
300 ON PEEK(753)<>3 GOTO 300:GOSUB 6
    35:ON N=12 GOTO 305:N=0:RETURN
305 N=1:RETURN
310 CLOSE #1:N=0:RETURN
315 ? "{CLEAR}{12 SPACES}Screenbyter"
    :? :? :?
320 GOSUB 695:? :? "What file should
     hold your finished{3 SPACES}scr
    een? (Eight characters)":POKE 76
    4,255:INPUT FSAVE$
325 FL$=FSAVE$:GOSUB 205:ON N GOTO 3
    20:FSAVE$=FLL$:FSAVE$(LEN(FLL$)+
    1)=".PIX"
330 FL$=FSAVE$:GOSUB 290:ON N GOTO 3
    20
335 FLOAD$="":? :? "Would you like t
    o edit a screen you{3 SPACES}hav
    e already saved? (Y or N) "
340 GOSUB 635:ON N=35 GOTO 390:ON N=
    43 GOTO 345:GOTO 340
345 ? :? "What is the name of the sa
    ved screen  file? ":POKE 764,255
    :INPUT FLOAD$
350 FL$=FLOAD$:GOSUB 205:ON N=0 GOTO
     355:GOTO 335
355 FLOAD$=FLL$:FLOAD$(LEN(FLL$)+1)=
    ".PIX"
360 FL$=FLOAD$:GOSUB 270:ON N GOTO 3
    35:OPEN #1,4,0,FLOAD$:GET #1,MD:
    GET #1,WD:GET #1,LLO:GET #1,LHI
365 CLOSE #1:FLOAD=1:LN=LLO+256*LHI
370 ? :? FLOAD$;" was saved as:":? "
    Mode ";MD;",":? "with ";LN;" lin
    es":? "of ";WD;" characters per
    line."
375 ? "If you wish to change these p
    arameterspress RETURN.":? "To le
    ave them unchanged press any
    {5 SPACES}other key."
380 ON PEEK(753)<>3 GOTO 380:GOSUB 6
    35:IF N=12 THEN 395
385 M=MD:M8=M-8:WIDE=WD:LINE=LN:GOTO
     445
390 FLOAD=0
```

```
395 ? :? "What Antic mode will you w
    ork in?":? :? "Antic","Graphics"
    :? 8,3:? 9,4:? "A (1Ø)",5:? "B (
    11)",6
4ØØ ? "C (12)",N$:? "D (13)",7:? "E
    (14)",N$:? "F (15)",8:POKE 764,2
    55
4Ø5 TRAP 4Ø5:OPEN #1,4,Ø,"K:":GET #1
    ,N:CLOSE #1:ON N<56 OR (N>57 AND
     N<65) OR N>7Ø GOTO 4Ø5
41Ø M=N-48:M=M-7*(M>9):M8=M-8
415 ? :? "How wide a line?":? "  (Mi
    nimum ";BPL(M8);" bytes":? "
    {3 SPACES}maximum ";MXW(M8);" by
    tes)"
42Ø POKE 764,255:TRAP 42Ø:INPUT WIDE
    :WIDE=INT(WIDE):GOSUB 64Ø:GOSUB
    745
425 ? :? "How many lines do you want
     to edit?":? "(Minimum ";LPS(M8)
    ;", Maximum ";MXL;")"
43Ø TRAP 43Ø:INPUT LINE
435 LINE=INT(LINE):ON LINE<=MXL AND
    LINE>=LPS(M8) GOTO 44Ø:LINE=MXL*
    (LINE>MXL)+LPS(M8)*(LINE<LPS(M8)
    )
44Ø LHI=INT(LINE/256):LLO=LINE-256*L
    HI
445 ? "{CLEAR}":? "You have chosen:"
    :? "Save file--";FSAVE$:? "Load
    file--";FLOAD$
45Ø ? "Mode ";M:? LINE;" lines of ";
    WIDE;" characters"
455 ? "If this is right, press START
    {9 SPACES}To make changes, press
     OPTION"
46Ø ON (PEEK(53279)=6)+(2*(PEEK(5327
    9)=3)) GOTO 465,315:GOTO 46Ø
465 ? "{CLEAR}Just a minute while I
    get myself{6 SPACES}together . .
     ."
47Ø SC=SC+((LINE*WIDE)>K4)*(K4-INT(K
    4/WIDE)*WIDE):SH=INT(SC/256):SL=
    SC-256*SH
475 POKE 167Ø,WIDE-1:POKE 1674,LLO-1
    +256*(LLO=Ø):POKE 1675,LHI-(LLO=
    255)
```

```
480 POKE 1678,BPL(M8)-1:POKE 1680,LP
    S(M8)-1:POKE 1692,CL(3):POKE 169
    3,PPB(M8):POKE 1698,LPS(M8):POKE
     1699,WIDE
485 GOSUB 755:GOSUB 490:GOSUB 505:GO
    SUB 650:GOSUB 530:ON FLOAD GOSUB
     500:GOTO 110
490 OPEN #1,4,0,"D1:CLEARS.SUB":FOR
    I=1 TO 26:GET #1,N:CLEAR$(I,I)=C
    HR$(N):NEXT I:CLOSE #1
495 C=USR(ADR(CLEAR$),SP):RETURN
500 T=SZ:FL$=FLOAD$:GOSUB 40:SZ=T:RE
    TURN
505 DL4=DL+4:DL5=DL+5:FOR I=0 TO 2:P
    OKE DL+I,112:NEXT I:C=INT(SC/256
    ):N=SC-C*256
510 FOR I=1686 TO 1688 STEP 2:POKE I
    ,N:POKE I+1,C:NEXT I:N=0
515 FOR I=DL+3 TO DL+3*LPS(M8) STEP
    3:C=SC+N*WIDE:POKE I,64+M:T=INT(
    C/256)
520 POKE I+2,T:POKE I+1,C-256*T:N=N+
    1:NEXT I
525 POKE I,65:POKE I+1,0:POKE I+2,DL
    /256:RETURN
530 OPEN #1,4,0,"D:SCROLL.SUB":N=INT
    (SCROLL/256):C=SCROLL-256*N
535 POKE ICBADR+X+1,N:POKE ICBADR+X,
    C:POKE ICBLEN+X+1,3:POKE ICBLEN+
    X,0
540 POKE ICCOM+X,7:I=USR(FMS,X):CLOS
    E #1
545 POKE 560,0:POKE 561,DL/256:CLOSE
     #1:RETURN
550 POKE PEEK(1688)+256*PEEK(1689),P
    EEK(1690):GOSUB 60:GRAPHICS 0:PO
    KE 764,255
555 ? "Screen is saved as D1:TEMPFIL
    E.SCR":? :? "Do you want to save
     the screen as":? FSAVE$;"? (Y o
    r N)"
560 GOSUB 635:ON N<>43 AND N<>35 GOT
    O 560:IF N=43 THEN GOSUB 610:GOT
    O 570
565 FSAVE=0
570 ? :? "Do you want to quit? (Y or
     N)":POKE 764,255
575 GOSUB 635:ON N<>43 AND N<>35 GOT
    O 575:ON N=35 GOTO 580:ON N=43 G
    OTO 605
```

```
580 ? :? "To return to edit the same
     screen,{4 SPACES}press OPTION":
    ? :? "To start SCREENBYTER over,
     press START"
585 OPT=PEEK(53279):ON ((OPT=6)+(2*(
    OPT=3))) GOTO 590,595:GOTO 585
590 POKE 106,A:GRAPHICS 0:GOTO 20
595 POKE 106,TOP:FL$="D1:TEMPFILE.PI
    X":IF FSAVE=1 THEN FL$=FSAVE$
600 GOSUB 755:GOSUB 40:GOSUB 505:POK
    E 560,0:POKE 561,DL/256:GOTO 110
605 POKE 106,A:POKE 764,255:GRAPHICS
     0:END
610 FSAVE=1:TRAP 615:OPEN #2,4,0,FSA
    VE$:CLOSE #2:XIO 36,#2,0,0,FSAVE
    $:XIO 33,#2,0,0,FSAVE$:GOTO 620
615 CLOSE #2
620 FL$="D1:TEMPFILE.PIX,":FLL$=FSAV
    E$(4,LEN(FSAVE$)):FL$(17)=FLL$
625 XIO 32,#1,0,0,FL$:RETURN
630 ON PEEK(753)<>3 GOTO 630:RETURN
635 C=PEEK(764):N=C-64*INT(C/64):RET
    URN
640 IF WIDE>=BPL(M8) AND WIDE<=MXW(M
    8) THEN RETURN
645 WIDE=MXW(M8)*(WIDE>MXW(M8))+BPL(
    M8)*(WIDE<BPL(M8)):RETURN
650 OPEN #1,4,0,"D:DELETES.SUB":FOR
    I=1 TO 118:GET #1,N:DELETE$(I,I)
    =CHR$(N):NEXT I:CLOSE #1
665 OPEN #1,4,0,"D:EXPANDS.SUB":FOR
    I=1 TO 102:GET #1,N:EXPAND$(I,I)
    =CHR$(N):NEXT I:CLOSE #1
680 OPEN #1,4,0,"D:FILL.SUB":FOR I=1
     TO 230:GET #1,N:FILL$(I,I)=CHR$
    (N):NEXT I
690 CLOSE #1:C=ADR(FILL$):N=INT(C/25
    6):C=C-N*256:POKE 1702,76:POKE 1
    703,C:POKE 1704,N:RETURN
695 TRAP 710:XIO 36,#1,0,0,"D1:*.PIX
    "
700 ? :? "Currently saved screen fil
    es:"
705 FL$="D1:*.PIX":OPEN #1,6,0,FL$:F
    OR I=0 TO 50:INPUT #1,FLL$:? FLL
    $:NEXT I
710 CLOSE #1:RETURN
715 FOR I=0 TO 10:POKE 53279,4:NEXT
    I:RETURN
```

```
720 SHIF=(C=31)+2*(C=30)+3*(C=26)
725 POKE 53279,4:POKE 1692,CL(SHIF):
    IF PPB(M8)=1 AND SHIF>0 THEN SHI
    F=3:POKE 1692,CL(SHIF)
730 RETURN
735 N=(VERS=255):SOUND 0,200*N,14*N,
    4*N:RETURN
740 FOR I=0 TO 10:NEXT I:RETURN
745 IF BPL(M8)=MXW(M8) THEN MXL=LPS(
    M8):RETURN
750 MXL=INT(K4/WIDE):RETURN
755 FOR I=1677 TO 1681 STEP 2:POKE I
    ,0:NEXT I:FOR I=1686 TO 1688 STE
    P 2:POKE I,SL:POKE I+1,SH:NEXT I
760 N=128+64*(PPB(M8)=2):POKE 1696,N
    :POKE 1701,N
765 POKE 1672,0:POKE 1676,0:VERS=0:G
    OSUB 735:POKE 1695,VERS:RETURN
770 DATA 2,10,170,24,1,10,85,48,2,20
    ,85,48,1,20,42,96
775 DATA 1,20,21,192,2,40,42,96,2,40
    ,40,192,1,40,40,192
```

Program 4. Insert Line Routine

```
900 OPEN #1,8,0,"D1:EXPANDS.SUB"
910 FOR I=1 TO 102:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 104,56,165,205,237,163,6,1
     33
1008 DATA 203,165,206,233,0,133,204,
     56
1016 DATA 173,138,6,237,140,6,133,20
     7
1024 DATA 173,139,6,237,141,6,133,20
     8
1032 DATA 165,208,240,5,162,255,24,1
     44
1040 DATA 2,166,207,172,163,6,177,20
     3
1048 DATA 145,205,136,208,249,202,24
     0,31
1056 DATA 56,165,205,237,163,6,133,2
     05
1064 DATA 165,206,233,0,133,206,56,1
     65
1072 DATA 203,237,163,6,133,203,165,
     204
```

```
1080 DATA 233,0,133,204,24,144,212,1
     65
1088 DATA 208,208,206,172,163,6,169,
     0
1096 DATA 145,203,136,208,251,96
```

Program 5. Delete Line Routine

```
900 OPEN #1,8,0,"D1:DELETES.SUB"
910 FOR I=1 TO 118:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 104,56,173,152,6,237,136,6
1008 DATA 133,203,173,153,6,233,0,13
     3
1016 DATA 204,24,165,203,109,163,6,1
     33
1024 DATA 205,165,204,105,0,133,206,
     56
1032 DATA 173,138,6,237,140,6,133,20
     7
1040 DATA 173,139,6,237,141,6,133,20
     8
1048 DATA 165,208,240,5,162,255,24,1
     44
1056 DATA 2,166,207,172,163,6,177,20
     5
1064 DATA 145,203,136,208,249,202,24
     0,31
1072 DATA 24,165,205,109,163,6,133,2
     05
1080 DATA 165,206,105,0,133,206,24,1
     65
1088 DATA 203,109,163,6,133,203,165,
     204
1096 DATA 105,0,133,204,24,144,212,1
     65
1104 DATA 208,208,206,172,163,6,169,
     0
1112 DATA 145,205,136,208,251,96
```

Program 6. Cursor Movement Routine

```
900 OPEN #1,8,0,"D1:SCROLL.SUB"
910 FOR I=1 TO 650:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 104,104,104,141,161,6,173,
     152
1008 DATA 6,133,207,173,153,6,133,20
     8
```

```
1016 DATA 160,0,140,158,6,173,154,6
1024 DATA 145,207,173,161,6,41,8,240
1032 DATA 92,173,161,6,41,4,208,71
1040 DATA 172,157,6,173,160,6,42,176
1048 DATA 8,136,208,250,141,160,6,24
     0
1056 DATA 54,42,136,208,252,141,148,
     6
1064 DATA 173,136,6,208,2,240,40,173
1072 DATA 148,6,141,160,6,56,173,136
1080 DATA 6,233,1,141,136,6,56,173
1088 DATA 152,6,233,1,141,152,6,173
1096 DATA 153,6,233,0,141,153,6,173
1104 DATA 143,6,240,6,206,143,6,24
1112 DATA 144,99,173,158,6,9,8,141
1120 DATA 158,6,24,144,88,172,157,6
1128 DATA 173,160,6,106,176,8,136,20
     8
1136 DATA 250,141,160,6,240,71,106,1
     36
1144 DATA 208,252,141,148,6,173,136,
     6
1152 DATA 205,134,6,208,2,240,54,173
1160 DATA 148,6,141,160,6,24,173,136
1168 DATA 6,105,1,141,136,6,24,173
1176 DATA 152,6,105,1,141,152,6,173
1184 DATA 153,6,105,0,141,153,6,173
1192 DATA 143,6,205,142,6,240,6,238
1200 DATA 143,6,24,144,8,173,158,6
1208 DATA 9,4,141,158,6,173,161,6
1216 DATA 41,1,240,83,173,161,6,41
1224 DATA 2,208,62,173,140,6,205,138
1232 DATA 6,208,8,173,141,6,205,139
1240 DATA 6,240,124,24,173,140,6,105
1248 DATA 1,141,140,6,173,141,6,105
1256 DATA 0,141,141,6,24,173,152,6
1264 DATA 109,163,6,141,152,6,173,15
     3
1272 DATA 6,105,0,141,153,6,173,145
1280 DATA 6,205,144,6,240,6,238,145
1288 DATA 6,24,144,75,173,158,6,9
1296 DATA 1,141,158,6,24,144,64,173
1304 DATA 140,6,208,5,173,141,6,240
1312 DATA 54,56,173,140,6,233,1,141
1320 DATA 140,6,173,141,6,233,0,141
1328 DATA 141,6,56,173,152,6,237,163
1336 DATA 6,141,152,6,173,153,6,233
1344 DATA 0,141,153,6,173,145,6,240
1352 DATA 6,206,145,6,24,144,8,173
```

```
1360 DATA 158,6,9,2,141,158,6,173
1368 DATA 152,6,133,207,173,153,6,13
     3
1376 DATA 208,173,132,2,240,36,160,0
1384 DATA 177,207,141,154,6,73,255,4
     5
1392 DATA 160,6,141,155,6,173,160,6
1400 DATA 73,255,45,154,6,13,155,6
1408 DATA 141,155,6,173,158,6,240,40
1416 DATA 208,41,160,0,177,207,141,1
     55
1424 DATA 6,173,156,6,45,160,6,141
1432 DATA 161,6,173,160,6,73,255,45
1440 DATA 155,6,141,155,6,13,161,6
1448 DATA 141,154,6,173,158,6,208,3
1456 DATA 24,144,98,41,8,240,17,56
1464 DATA 173,150,6,233,1,141,150,6
1472 DATA 173,151,6,233,0,141,151,6
1480 DATA 173,158,6,41,4,240,17,24
1488 DATA 173,150,6,105,1,141,150,6
1496 DATA 173,151,6,105,0,141,151,6
1504 DATA 173,158,6,41,1,240,18,24
1512 DATA 173,150,6,109,163,6,141,15
     0
1520 DATA 6,173,151,6,105,0,141,151
1528 DATA 6,173,158,6,41,2,240,24
1536 DATA 56,173,150,6,237,163,6,141
1544 DATA 150,6,173,151,6,233,0,141
1552 DATA 151,6,24,144,3,24,144,67
1560 DATA 173,150,6,133,203,173,151,
     6
1568 DATA 133,204,24,173,48,2,105,4
1576 DATA 133,205,173,49,2,133,206,1
     74
1584 DATA 162,6,160,0,165,203,145,20
     5
1592 DATA 200,165,204,145,205,24,165
     ,205
1600 DATA 105,3,133,205,165,206,105,0
1608 DATA 133,206,24,165,203,109,163
     ,6
1616 DATA 133,203,165,204,105,0,133,
     204
1624 DATA 202,208,215,173,155,6,160,
     0
1632 DATA 145,207,173,159,6,201,255,
     208
1640 DATA 8,173,132,2,208,3,32,166
1648 DATA 6,96
```

Program 7. Clear Screen Routine

```
900 OPEN #1,8,0,"D1:CLEARS.SUB"
910 FOR I=1 TO 26:READ N:PUT #1,N:NE
    XT I:CLOSE #1:? I:END
1000 DATA 104,104,104,133,208,162,32
     ,169
1008 DATA 0,133,207,160,255,145,207,
     136
1016 DATA 208,251,145,207,230,208,20
     2,208
1024 DATA 238,96
```

Program 8. Fill Subroutine

```
900 OPEN #1,8,0,"D1:FILL.SUB"
910 FOR I=1 TO 230:READ N:PUT #1,N:N
    EXT I:CLOSE #1:? I:END
1000 DATA 173,136,6,141,135,6,173,15
     4
1008 DATA 6,141,146,6,165,207,133,20
     3
1016 DATA 165,208,133,204,162,0,173,
     160
1024 DATA 6,141,148,6,172,157,6,78
1032 DATA 148,6,176,52,136,208,248,1
     73
1040 DATA 146,6,45,148,6,141,149,6
1048 DATA 173,156,6,45,148,6,205,149
1056 DATA 6,240,20,141,149,6,173,148
1064 DATA 6,73,255,45,146,6,13,149
1072 DATA 6,141,146,6,24,144,205,173
1080 DATA 146,6,129,207,141,154,6,96
1088 DATA 173,135,6,205,134,6,240,24
     7
1096 DATA 238,135,6,24,165,203,105,1
1104 DATA 133,203,165,204,105,0,133,
     204
1112 DATA 161,203,141,148,6,173,165,
     6
1120 DATA 141,164,6,173,164,6,45,148
1128 DATA 6,141,149,6,173,164,6,45
1136 DATA 156,6,205,149,6,240,13,172
1144 DATA 157,6,78,164,6,176,193,136
1152 DATA 208,248,240,223,172,157,6,
     14
1160 DATA 164,6,176,29,136,208,248,1
     73
1168 DATA 164,6,45,156,6,141,149,6
```

```
1176 DATA 173,164,6,73,255,45,148,6
1184 DATA 13,149,6,141,148,6,24,144
1192 DATA 219,162,Ø,173,148,6,129,2Ø
     3
12ØØ DATA 173,146,6,129,2Ø7,141,154,
     6
12Ø8 DATA 56,173,135,6,237,136,6,24Ø
1216 DATA 12,168,136,24Ø,8,173,156,6
1224 DATA 145,2Ø7,136,2Ø8,251,96
```

6

Advanced Techniques

Moving the Display

Michael P. Surh

A variation on the techniques which allow screen flipping also makes possible screen scrolling—even horizontally. Here's how.

The Atari computers are capable of moving their display memory (that is, the section of memory storing the screen display), and they can have more than one section of display memory stored at the same time. This feature is particularly useful for graphics displays and can be used for animation. It is possible to move the entire screen so that everything you see on the screen moves too, and the technique can be used to create smoother animation and drawing than would otherwise be possible.

Before you try the following programs which demonstrate these tricks, you should have some understanding of how the Atari display works. The Atari has two separate registers that control where it keeps its display (the display list) in the overall computer memory. These registers work by storing the address of the first memory location used by the display list. The computer puts all of the screen data into the memory starting at that first address, and also reads the numbers stored there to display images on the screen. Since what you see on the screen is stored in this memory section, it is possible to modify the display directly by using POKEs instead of the usual BASIC commands PRINT, PLOT, and DRAWTO.

If this is news to you, try Program 1 to modify a GRAPHICS 0 screen with POKEs. This program begins by printing the address for the start of the display list as read from both of the registers (see lines 20 through 50). Then the program POKEs numbers from 0 to 255 into successive locations in the display list. All of the alphanumeric and graphics characters appear on the screen. Also, try changing the graphics mode on line 10 (but remove line 40 if the graphics mode is not mixed with a GRAPHICS 0 text window). This program works on any graphics mode, but its effects are different with various modes. At times this graphics technique is better than using PRINT or PLOT and DRAWTO because it is faster, even though it is more difficult.

The program works because the computer keeps the display of characters or graphics points stored as numbers in the reserved display memory. Each number stored there corresponds to a character or some graphics points on the screen. Whenever the computer prints or draws something, it is going into this memory section and changing something (this is what Program 1 did, without using PRINT or DRAWTO). Changing the display list changes the screen because the computer also reads this memory section from start to finish and sends it to the screen 60 times every second.

Controlling Display Memory

The two registers that store the location of the display list tell the computer where to read from or write to the display memory; without them, the computer would not be able to find it. There is a good reason for storing the display address in two locations—this allows sophisticated graphics and animation. Both of the registers act as pointers to the display memory, and they both store its starting address, but one pointer controls where the computer goes to write in the memory, and the other tells the computer where to start reading the information to be displayed. If you remember, the computer must do both operations to put the display on your screen, and it has a pointer for each function.

The pointer controlling all writing to the screen is located at 88 and 89 in the memory, but the second pointer's location depends on your computer's memory size and the graphics mode you are using. See Program 1, line 20, to find out how to locate the pointer; its position is the variable PNTR and PNTR + 1.

Usually the two pointers store the same address, so if the computer prints something on the third line, the information appears on the third line of the screen in exactly the same place as it was written in the display list. But if you change one of the two pointers, when the computer wants to read or write on the display memory and goes to what it thinks is the start of the memory section, it is in the wrong location. As a result, the information appears in the wrong place.

Program 2 demonstrates what happens when you change the pointer controlling where the computer writes into the memory. The program starts by printing the word *left* all the way down the left margin of the screen. Then it clears the screen and starts over, but this time it changes the number stored in 88 each time it prints a word. The computer still prints on the left margin (at least it thinks it does!), but the words are stored in the memory offset from their correct positions, so they appear in the wrong places on the screen.

Once the computer is given the wrong starting address, everything it prints afterward will appear in the wrong place. This is because the computer starts at the location specified by the register in 88 and 89 and counts through the memory until it is where it wants to print. If it starts in the wrong place, it winds up in the wrong place, and whatever is printed or drawn is in a different spot on the screen. If you want to prove that the computer is starting in the wrong place, hit BREAK (not SYSTEM RESET). The READY prompt and anything you type will line up at the new margin, and the lines will overlap onto both sides of the screen.

Notice that whatever was written on the screen before location 88 was changed did not move; only the words printed after the change are displaced. Also, it is not always possible to print at the bottom of the screen, and it sometimes becomes impossible to use the PLOT function after the numbers in 88 or 89 have been changed. Since I can find no way to remedy these problems, I see little use for this even though it is an interesting trick.

The other pointer tells the computer where to start *reading* the display memory. Changing the address stored here is much more interesting, because it makes the computer start reading in the wrong place, and the entire screen shifts. By controlling how much the pointer changes from its original value, you can make everything on the screen seem to move horizontally or vertically.

Unfortunately, it is not very practical to move the screen vertically because garbage is sometimes displayed. You can see this in Program 3, which changes the pointer to move any graphics mode horizontally. This pointer is stored in a variable because it is placed just before the start of the display memory, and the display memory's location depends on the graphics mode and your computer's memory size.

Moving the Screen

When you try this program, you may notice certain features of the moving screen. First, in any of the mixed graphics modes, the text window of GRAPHICS 0 at the bottom of the screen remains stationary as the rest of the screen moves. In GRAPHICS 8 and 8 + 16, only the top half of the screen moves while the bottom is at rest. Also, the screen jumps each time the loop is reexecuted, and as the program runs through the loop, part of the screen fills with apparently random data (garbage). Last of all, the edge of the screen that moves out of view horizontally reappears on the other side of the screen.

The text window of GRAPHICS 0 in the mixed graphics modes does not move because it has its own pointer to control the start of its display memory. This is also true for the bottom half of a GRAPHICS 8 or 8 + 16 display. Check Table 1 to find where to POKE to move the bottom half of GRAPHICS 8 and the text window for each of the mixed graphics modes.

There is an advantage to this added complexity. Not only can you move part of the screen and leave the rest still, but you can also move the different parts in different directions or at different rates. In GRAPHICS 8 you can actually move all three parts at different speeds at the same time.

That cures the problem of unmoving parts of the screen, but there are still more problems. When the program finishes its loop and starts over again, there is a large and noticeable jump on the screen. Also, there are unusual problems with the top of the screen; unwanted garbage occasionally appears or part of the display disappears off the top. This is particularly noticeable in GRAPHICS 0 through 2, which are text modes. Strange characters can appear, and if you erase them the display goes haywire.

You can reduce the jump in the screen each time the loop is run. Change the loop in line 70 to match what is in Table 2 for the particular graphics mode. This also remedies the occasional appearance of mysterious characters at the top of the screen. Unfortunately, this means that the top line will periodically disappear and reappear. You could leave it blank to keep this unnoticed.

Still, the method is satisfactory for the higher resolution graphics modes where the screen "bumps" are less obvious. And by carefully adapting the loop, you might find a decent compromise.

Try the programs included with this article, and experiment with different graphics modes to get an idea of the possibilities and limitations of these unusual features.

Table 1. Where to POKE to Move Parts of the Screen

To Move GR.0 Text Window in:	POKE into PNTR−4 +
GR.1	26 or 27
GR.2	16 or 17
GR.3	26 or 27
GR.4	46 or 47
GR.5	46 or 47
GR.6	86 or 87
GR.7	86 or 87
GR.8	168 or 169

To move lower part of GR.8, POKE into PNTR−4 + 100 or 101.

Table 2. Smoothing the Horizontal Motion of the Screen

Graphics Mode	Change Line 70 to FOR LOOP=
0	X TO X + 39
1	X TO X + 19
2	X TO X + 19
3	X TO X + 9
4	X TO X + 9
5	X TO X + 19
6	X TO X + 19
7	X TO X + 39
8	X TO X + 39

Program 1. Display Using POKE

```
0 REM ******* PROGRAM1 *******
10 GRAPHICS 0
20 SCR1=PEEK(88)+256*PEEK(89)
30 PNTR=PEEK(560)+256*PEEK(561)+4
35 SCR2=PEEK(PNTR)+256*PEEK(PNTR+1)
40 ? SCR1,SCR2
45 FOR PAUSE=1 TO 500:NEXT PAUSE
50 FOR LOOP=0 TO 255
60 POKE SCR2+LOOP,LOOP
70 NEXT LOOP
80 GOTO 80
```

Program 2. Changing the Pointer to Screen Memory

```
0 REM PROGRAM 2
10 GRAPHICS 0
20 FOR LOOP=1 TO 20
30 ? "LEFT"
40 NEXT LOOP
50 FOR PAUSE=1 TO 500:NEXT PAUSE
60 ? CHR$(125)
70 FOR LOOP=1 TO 20
80 ? "LEFT"
90 POKE 88,PEEK(88)+1
100 NEXT LOOP
110 GOTO 110
```

Program 3. Moving Horizontally

```
0 REM PROGRAM 3
10 GRAPHICS 0
20 FOR LOOP=1 TO 100
30 POSITION RND(1)*30,RND(1)*20:? "*"
   :REM FILL SCREEN WITH ASTERISKS
40 NEXT LOOP
50 PNTR=PEEK(560)+256*PEEK(561)+4:REM
    LOCATION OF READ POINTER
60 X=PEEK(PNTR):REM STARTING POINT OF
    SCREEN
70 FOR LOOP=0 TO 255
80 POKE PNTR,LOOP:REM CHANGE ADDRESS
   IN POINTER
90 FOR PAUSE=1 TO 10:NEXT PAUSE
100 NEXT LOOP
110 GOTO 70
```

Screen Flipping

Michael Kirtley

Here's another way to rapidly "flip" screens.

This useful subroutine can be appended to your own BASIC programs. It lends them a cleaner, more professional look. The screen-flipping routine can be used to include a page or two of documentation with your program or to switch to a menu instantly. Another intriguing use would be to flip rapidly between several screens to create animation.

When you first run the program, there will be a brief pause as the Atari writes screen two. Then you will see screen one being written. I intentionally left this in to contrast between writing a screen and flipping to a previously written one. This effect can be eliminated by POKEing 559 with a 0 to turn off the video display until the screen has been written. Another advantage in using this POKE is a significant increase in execution speed. You can try this by adding the following lines to the program:

```
130 R=PEEK(559):POKE 559,0
180 POKE 559,R
```

Another embellishment to the program would be to write screen one and show it to the user. While the user is viewing screen one, the computer could be writing screens two, three, etc.

A small "beep" sound subroutine placed just before the POKEs on lines 220 and 230 will give a crisper effect upon flipping screens. I've found the following routine to be a good one for this purpose.

```
250 SOUND 0,100,10,15:FOR X=1 TO 20:NEXT X
260 SOUND 0,0,0,0:RETURN
```

Now let's look at how the program works.

Lines 100-120: Initialize variables. Locations 560 and 561 hold the low and high bytes of the address of the display list. Location 742 is the high byte of the top of memory pointer. Location 89 is the high byte of the lowest address of screen memory.

Line 140: Tell the Atari where to write screen two. Before writing screen two, clear the screen.

Line 150: Write screen two. You can substitute your own application here.

Line 160: Tell the Atari where to write screen one.

Line 170: Write screen one.
Line 190: GET a keyboard response from the user.
Line 200: GOTO whichever screen the user selects.
Line 210: Go back and get another keyboard response if the user picks a screen we don't have.
Line 220: POKE the location of screen one into the correct place in the display list (DL + 5). This is where the Atari looks to find where the current screen is located.
Line 230: POKE the location of screen two into the display list.

Additional screens can be added (depending on memory size and the application program) by stepping down four 256-byte pages from the location of the previous screen. In graphics mode 0 there are 24 lines with 40 one-byte characters each, for a total of 960 bytes. This is a little less than four pages (1024 bytes). Therefore, if

SCREEN2 = MEMTOP − 5	**then**
SCREEN3 = MEMTOP − 9	**and**
SCREEN4 = MEMTOP − 13	**etc.**

To make room for screen two, you must step down five pages from MEMTOP to avoid garbage on the screen. To add additional screens, initialize as above, then POKE 89 with the new screen and POKE 106 with the new screen + 4. You will now be able to write the new screen. To view the screen, just POKE DL + 5 with the new screen.

Now enter the program and run it. Press 1 for screen one and 2 for screen two.

Screen Flipping

```
100 DL=PEEK(560)+256*PEEK(561)
110 MEMTOP=PEEK(742)
120 SCREEN1=PEEK(89):SCREEN2=MEMTOP-5
140 POKE 89,SCREEN2:POKE 106,SCREEN2+
    4:? CHR$(125)
150 FOR X=1 TO 22:? "********* THIS I
    S SCREEN 2 *********":NEXT X
160 POKE 89,SCREEN1:POKE 106,SCREEN1+
    4:? CHR$(125)
170 FOR X=1 TO 22:? "-------- SCREEN
    1 GOES HERE --------":NEXT X
190 OPEN #1,4,0,"K:":GET #1,X:CLOSE #
    1:X=X-48:IF X<1 THEN 190
200 ON X GOTO 220,230
210 GOTO 190
220 POKE DL+5,SCREEN1:GOTO 190
230 POKE DL+5,SCREEN2:GOTO 190
```

Artifacting

Judson Pewther

These tools for exploring artifacting can create some of the most beautiful graphics you've ever seen from your Atari.

Even if you are not already familiar with the phrase *television artifacts,* you have probably noticed that the colors of points and lines drawn in Atari's graphics mode 8 are not always what they are supposed to be. (False colors may also appear in graphics mode 0.)

Although the *BASIC Reference Manual* claims that only "one color and two different luminances" are available in GRAPHICS 8, in fact six distinguishable color/luminance combinations are possible because of TV artifacting.

While the *BASIC Reference Manual* does not mention this very interesting fact, it is fully documented in *De Re Atari,* Appendix IV, which gives the definition: "The term TV artifacts refers to a spot or 'pixel' on the screen that displays a different color than the one assigned to it." And, as further explained, TV artifacts are caused by the way in which color and luminance information is modulated onto an NTSC television signal.

Let's summarize the effects of artifacting in GRAPHICS 8:

1. The effect is maximized by plotting a light color (high luminance) on a dark background, or dense dark patterns on a light background.
2. The color of a pixel is not affected by its Y-coordinate.
3. The color displayed by a pixel depends not only on its assigned color, but also on whether its X-coordinate is even or odd, and on the color assigned to its horizontal neighbors.
4. Horizontal resolution has a practical limit of 160 rather than 320. Thus, two horizontally contiguous pixels tend to form a single pixel of uniform color.

What colors are actually produced? This can depend on the particular TV monitor being used and on the exact setting of its controls. The setting of the tint control will make the biggest difference.

The major effects of plotting white (the assigned color) pixels on a black background are summarized in the following table. N is the number of horizontally contiguous white pixels. X is the X-coordinate(s) of these pixels in terms of "even" and "odd." COLOR is the approximate actual color displayed by these pixels, assuming normal settings on the TV monitor.

N	X	Color
1	even	green
1	odd	blue
2	even-odd	orange
2	odd-even	light blue
3	—	nearly white
4	—	white

The Table Illustrated

Program 1 illustrates artifacts by drawing two series of nearly vertical white lines on a black background. Colored horizontal bands are produced in accordance with the rules in the previous table. No actual white is produced in this example, because there are at most only pairs of horizontally contiguous "white" pixels. Notice in particular that the solid-color bands are created either because all the "even" pixels give a solid green, or all the "odd" pixels give a solid blue.

Lines 199 to 250 can be added to allow the user to easily step the assigned hue through all 16 possibilities, while preserving the 0 luminance setting for the background and the 14 luminance setting for the plotted lines. The background color may be nearly invisible because it is at 0 luminance, but the colors in the horizontal bands will change greatly. Remember that in GRAPHICS 8 the hue associated with the COLOR 1 statement and with the lines that were drawn is the background hue as determined in the SETCOLOR 2,hue,0 statement. Even when we are not seeing the assigned hue because of TV artifacting, changing the assigned hue changes the displayed hues.

Best results are obtained by adjusting TV brightness and contrast to a low or minimum value. TV color may be boosted somewhat, but too much boost blurs the picture. However, the tint control may be adjusted freely from one extreme to the other to vary the colors. These comments apply generally to any program where TV artifacts are used.

TV artifacts are really a failure of resolution, but a very interesting failure. And the colors produced can add dazzle to graphics art programs. Although these false colors may at times be annoying, and although the failure in horizontal resolution is certainly an annoyance, TV artifacts compensate considerably for the fact that only two intensities of a single color are officially available in GRAPHICS 8.

Moiré Patterns

Program 2 is a graphics art program which relies on artifacts for its beauty. It also makes use of a technique for creating enhanced moiré patterns.

To see a somewhat different moiré pattern with a more uniform distribution of light and enhanced contrasts in the details, add the following line and run the program again:

```
50 COLOR 0:PLOT 159,0:DRAWTO X+1,191
```

This program step draws a black line which cancels out half (or more) of the "white" pixels which were plotted in the previous step, line 40. This basic idea is varied and elaborated in Program 3, "Pyramid."

Program 3 is designed so that slow typists (like myself) will not have to type in the whole thing just to see what it does. The first half of the program (lines 100 through 470) is almost entirely for the purpose of letting the user control the parameters of the pattern in order to see better how the various effects are achieved. To eliminate some typing, replace the first half of the program with the single line: 100 GRAPHICS 24. Then begin typing at line 500.

The program is essentially self-explanatory, but it might be worthwhile to point out a few things. Lines 500 to 545 select a set of random parameters for the pattern that is about to be drawn. LIGHT and DARK are associated with the subroutine for drawing a set of vertical lines at line 1000 in the program. They are dual purpose variables: if equal to 0 or 1, then a set of "even" or "odd" lines will be drawn, but if greater than 1 the subroutine will not be called. So the probability is .25 that LIGHT will call the subroutine, since it is a random integer ranging from 0 to 7. The same applies to DARK.

LIGHT lays down a colored background for the pattern, but has a slightly different effect if the old pattern has not been wiped out by line 730. DARK erases all colors in the pattern except for black and another color, just before the program recycles to select a new set of random parameters.

Line 535 works in conjunction with line 740 to insure that the new values of MODE, APEX, and SPACE are not exactly the same as the old values.

Line 550 prevents the attract mode from setting in as long as the program continues to recycle through new variations.

Except for the user option to hold a pattern indefinitely (lines 450 and 720), there are no forced time delays. It takes about a minute for the program to make one cycle, which should be more than enough time to observe a variation of the pattern. If you wish to freeze a particular pattern, program execution may be stopped and restarted by hitting CTRL 1.

Program 4 illustrates TV artifacts by way of a dense pattern of dark lines drawn on a light background. Equally spaced rays are drawn from each of the four corners of a square to the opposite two sides. The light-colored pixels left over after this process is finished form a cross made out of four trumpet-shaped segments, corresponding roughly to what my dictionary identifies as a "formée" cross. The display is quite dazzling if one first adjusts for a pleasing color combination, and then turns out the lights in the room.

Program 1. Artifacts

```
10 GRAPHICS 8:COLOR 1
20 SETCOLOR 1,0,14:SETCOLOR 2,0,0
30 FOR X=0 TO 318 STEP 4
40 PLOT X,0:DRAWTO X+1,159:NEXT X
50 FOR X=0 TO 308 STEP 4
60 PLOT X,0:DRAWTO X+9,159:NEXT X
199 REM *** CHANGE HUE ***
200 OPEN #1,4,0,"K:"
210 ? :? "ASSIGNED HUE IS NOW ";H
220 ? "HIT H KEY TO CHANGE HUE"
230 GET #1,X:IF X<>72 THEN 230
240 H=H+1:IF H=16 THEN H=0
250 SETCOLOR 2,H,0:GOTO 210
```

Program 2. Simple Moiré

```
10 GRAPHICS 24
20 SETCOLOR 1,0,14:SETCOLOR 2,0,0
30 FOR X=0 TO 318 STEP 3
40 COLOR 1:PLOT 159,0:DRAWTO X,191
60 NEXT X
70 GOTO 70
```

Program 3. Pyramid

```
100 GRAPHICS Ø:POKE 82,5:POSITION 9,
    2:? "*** THE PYRAMID ***":PRINT
112 ? "ADJUST TV CONTRAST AND"
113 ? "BRIGHTNESS TO MINIMUM.":?
114 ? "ADJUST TV COLOR AND TINT"
115 ? "TO SUIT INDIVIDUAL TASTE.":?
12Ø ? "YOUR CHOICE:"
13Ø ? "   (Ø) RANDOM PARAMETERS"
14Ø ? "   (1) USER CONTROLLED PARAMET
    ERS"
15Ø TRAP 15Ø:INPUT CHOICE:IF CHOICE=
    Ø THEN GRAPHICS 24:GOTO 5ØØ
16Ø IF CHOICE<>1 THEN 15Ø
17Ø ? :? "DRAW BACKGROUND OF VERTICA
    L LINES?"
18Ø ? "   (Ø) EVEN LINES"
19Ø ? "   (1) ODD LINES"
2ØØ ? "   (2) NO BACKGROUND"
21Ø TRAP 21Ø:INPUT LIGHT:IF LIGHT=2
    THEN 23Ø
22Ø IF LIGHT<>Ø AND LIGHT<>1 THEN 21
    Ø
23Ø ? :? "DRAW PYRAMID IN MODE"
24Ø ? "   (Ø) FROM CENTER OUTWARD"
25Ø ? "   (1) FROM LEFT TO RIGHT"
26Ø TRAP 26Ø:INPUT MODE
27Ø IF MODE<>Ø AND MODE<>1 THEN 26Ø
28Ø ? :? "FOR APEX OF PYRAMID USE"
29Ø ? "   (Ø) ONE POINT"
3ØØ ? "   (1) TWO POINTS"
31Ø TRAP 31Ø:INPUT APEX
32Ø IF APEX<>Ø AND APEX<>1 THEN 31Ø
33Ø ? :? "SPACING OF RAYS FROM APEX?
    "
34Ø ? "(USUALLY AN INTEGER: 2 TO 6)"
35Ø TRAP 35Ø:INPUT SPACE:IF SPACE<1
    THEN 35Ø
36Ø ? :? "DRAW DARK VERTICAL LINES?"
37Ø ? "   (Ø) EVEN LINES"
38Ø ? "   (1) ODD LINES"
39Ø ? "   (2) NO LINES"
4ØØ TRAP 4ØØ:INPUT DARK:IF DARK=2 TH
    EN 42Ø
41Ø IF DARK<>Ø AND DARK<>1 THEN 4ØØ
42Ø ? :? "WHEN FINISHED"
43Ø ? "   (Ø) GOTO RANDOM PARAMETERS"
44Ø ? "   (1) HOLD THE PATTERN"
```

```
450 TRAP 45Ø:INPUT HOLD
455 IF HOLD=Ø THEN GRAPHICS 24:GOTO
    6ØØ
46Ø IF HOLD<>Ø AND HOLD<>1 THEN 45Ø
47Ø POKE 82,2:? :? "WHEN PATTERN IS
    FINISHED"
475 ? "HIT H KEY TO CHANGE HUE."
48Ø ? :? "READY?  HIT START KEY."
49Ø IF PEEK(53279)<>6 THEN 49Ø
495 TRAP 4ØØØØ:GRAPHICS 24:GOTO 6ØØ
497 REM
498 REM *** RANDOM PARAMETERS***
499 REM
5ØØ LIGHT=INT(RND(Ø)*8)
51Ø MODE=INT(RND(Ø)*2)
52Ø APEX=INT(RND(Ø)*2)
53Ø SPACE=2+INT(RND(Ø)*5)
535 IF MODE=M AND APEX=A AND SPACE=S
     THEN 51Ø
54Ø DARK=INT(RND(Ø)*8)
545 HUE=INT(RND(Ø)*16)
55Ø POKE 77,Ø:REM RESET ATTRACT MODE
597 REM
598 REM *** PROGRAM EXECUTION ***
599 REM
6ØØ SETCOLOR 2,HUE,Ø:SETCOLOR 1,Ø,14
62Ø IF LIGHT>1 THEN 64Ø
63Ø COLOR 1:B=LIGHT:GOSUB 1ØØØ
64Ø IF MODE=Ø THEN GOSUB 2ØØØ
65Ø IF MODE=1 THEN GOSUB 3ØØØ
66Ø REM *** DRAWTO SIDES ***
67Ø FOR K=191 TO 1 STEP -SPACE
68Ø COLOR 1:PLOT 159,APEX:DRAWTO 318
    ,K:PLOT 159,APEX:DRAWTO Ø,K
69Ø COLOR Ø:PLOT 159,Ø:DRAWTO 318,K-
    1:PLOT 159,Ø:DRAWTO Ø,K-1:NEXT K
7ØØ IF DARK>1 THEN 72Ø
71Ø COLOR Ø:B=DARK:GOSUB 1ØØØ
72Ø IF HOLD=1 THEN 4ØØØ
73Ø IF RND(Ø)<Ø.2 THEN GRAPHICS 24
74Ø M=MODE:A=APEX:S=SPACE
75Ø GOTO 5ØØ
998 REM
999 REM *** SUBROUTINE VERTICAL LINE
    S
1ØØØ FOR K=B TO 319 STEP 2
1Ø1Ø PLOT K,Ø:DRAWTO K,191:NEXT K
1Ø2Ø RETURN
1998 REM
1999 REM *** SUB STARTS FROM CENTER
```

```
2000 FOR K=0 TO 158 STEP SPACE
2010 COLOR 1:PLOT 159,0:DRAWTO 159+K
     ,191:PLOT 159,0:DRAWTO 159-K,19
     1
2020 COLOR 0:PLOT 159,APEX:DRAWTO 16
     0+K,191:PLOT 159,APEX:DRAWTO 15
     8-K,191
2030 NEXT K:RETURN
2998 REM
2999 REM *** SUB DRAWS LEFT TO RIGHT
3000 FOR K=0 TO 318 STEP SPACE
3010 COLOR 1:PLOT 159,APEX:DRAWTO K,
     191
3020 COLOR 0:PLOT 159,0:DRAWTO K+1,1
     91:NEXT K:RETURN
4000 REM
4001 REM *** ALLOWS USER TO CHANGE
4002 REM *** HUE WHEN PATTERN IS ON
4003 REM *** HOLD BY HITTING H KEY.
4010 OPEN #1,4,0,"K:"
4020 GET #1,X:IF X=72 THEN HUE=HUE+1
4030 IF HUE=16 THEN HUE=0
4040 SETCOLOR 2,HUE,0:GOTO 4020
```

Program 4. Cross

```
40 REM *** PROGRAM 4: JEWELED CROSS
50 GRAPHICS 0:POSITION 8,2:? "THE JE
   WELED CROSS":? :?
55 ? "WHEN PATTERN IS FINISHED--":?
60 ? "TURN ROOM LIGHTS DOWN TO GET"
70 ? "THE EFFECT OF A STAINED GLASS"
80 ? "WINDOW.":?
90 ? "HIT H KEY TO CHANGE HUE AND/OR
   "
95 ? "ADJUST TV TINT CONTROL.":?
100 ? "DIAMETER OF SQUARE?"
110 ? "(MAXIMUM OF 4*47--TRY 4*46)"
120 TRAP 150:? "4*";:INPUT DIAM:DIAM
    =4*DIAM
130 IF DIAM<1 OR DIAM>188 THEN 150
140 TRAP 40000:GOTO 300
150 PRINT CHR$(253):GOTO 50
300 REM *** DEFINE CORNERS OF SQUARE
310 XLO=INT(159.5-DIAM/2):XHI=XLO+DI
    AM
320 YLO=INT(95.5-DIAM/2):YHI=YLO+DIA
    M
349 REM
350 REM *** DRAW BACKGROUND ***
```

```
360 GRAPHICS 24:SETCOLOR 2,Ø,Ø:SETCO
    LOR 1,Ø,14:COLOR 1
41Ø FOR X=XLO TO XHI
42Ø PLOT X,YLO:DRAWTO X,YHI:NEXT X
449 REM
45Ø REM *** DRAW CROSS ***
46Ø COLOR Ø:FOR K=Ø TO DIAM STEP 4
47Ø PLOT XLO,YLO:DRAWTO XHI,YHI-K
48Ø PLOT XLO,YLO:DRAWTO XHI-K,YHI
49Ø PLOT XHI,YLO:DRAWTO XLO,YHI-K
5ØØ PLOT XHI,YLO:DRAWTO XLO+K,YHI
51Ø PLOT XHI,YHI:DRAWTO XLO,YLO+K
52Ø PLOT XHI,YHI:DRAWTO XLO+K,YLO
53Ø PLOT XLO,YHI:DRAWTO XHI,YLO+K
54Ø PLOT XLO,YHI:DRAWTO XHI-K,YLO
55Ø NEXT K
599 REM
6ØØ REM *** CHANGE HUE:HIT H KEY ***
61Ø OPEN #1,4,Ø,"K:"
62Ø GET #1,X:IF X=72 THEN HUE=HUE+1
63Ø IF HUE=16 THEN HUE=Ø
64Ø SETCOLOR 2,HUE,Ø:GOTO 62Ø
1ØØØ GOTO 1ØØØ
```

7

Protecting Memory

7

Positioning Player/Missile and Regular Graphics in Memory

Fred Pinho

Strange things indeed can appear on your display if memory is not protected properly for player/missile graphics. These charts should help you avoid memory conflicts in your Atari.

Have you ever used player/missile graphics only to notice funny-looking colored lines or dots on the screen with your carefully crafted images? When you moved your player or missile, these lines and dots seemed to acquire a life of their own. While it was fascinating to watch this "extra" display, it also quickly became frustrating to your programming attempts. The problem is that all the instructional articles I've seen tell you that you must step back in RAM a minimum of 1K (four pages) for double-line resolution and 2K (eight pages) for single-line resolution. They either ignore, or barely mention in passing, the important fact that you must also allow for the screen display memory in this calculation.

The Atari uses two blocks of memory to control the TV screen display. Residing at the very top of RAM is the display data. This block of memory contains a bitmap for the TV screen in graphics modes 3-8 and a character map for text modes 0-2. Residing just below the display data is the display list. This block of memory is essentially a short program that tells the Atari how to set up the TV screen for the desired mode. The total memory required for the display list and display data varies with the graphics mode used. This is illustrated in Table 1. As you can see, the highest resolution mode, GRAPHICS 8, requires the most RAM.

Thus, the explanation for the "extra bonus" lines or dots in

Table 1. Memory Required for Display List and Display Data

GRAPHICS MODE*	TOTAL MEMORY BYTES ALLOCATED TO						Total Bytes	Memory Step-Back To Be Added To P/M Step-Back, Pages †
	DISPLAY DATA				DISPLAY LIST			
	Bottom Text Window	Unused Bytes		Text or Graphics Screen	Unused Bytes	Used Bytes		
		Always	Conditional					
0	none	none	none	960	none	32	992	4
1	160	none	80	400	none	34	674	3
2	160	none	40	200	none	24	424	2
3	160	none	40	200	none	34	434	2
4	160	none	80	400	none	54	694	3
5	160	none	160	800	none	54	1174	5
6	160	none	320	1600	none	94	2174	9
7	160	none	640	3200	96	94	4190	17
8	160	16	1280	6400	80	176	8112	32

*If 16 is added to the GRAPHICS mode number, then the conditional unused bytes are added to the screen memory block. The bytes formerly used for the text window then become unused. Also the display list expands slightly.

†To be added to player/missile step-back pages.

your P/M display is that the program did not step back far enough into RAM and consequently located the P/M data in the display data memory area. The Atari then obediently displayed this data both from the normal display and through the P/M system. Since the display data is displayed as a number of bytes per line (Table 2), you will see a line of varying colored dots. By contrast, the P/M display is organized to display the bytes in a "stack" arrangement and so you see the desired figure (possibly as you designed it).

To aid you in using P/M graphics, Table 1 gives the number of pages that must be stepped back in memory (from the top of available RAM) to avoid interference between the two systems. For those not familiar with the concept of paging, the memory addressing system of the 6502 microprocessor within the Atari is based on the concept of a memory *page*. Each page is equivalent to 256 bytes of memory. Thus there are four pages of memory in each K (1024 bytes) of memory.

Note that, in calculating the step-back value for Table 1, a restriction must be observed: positioning for the P/M RAM must be on a 1K boundary for double-line resolution and on a 2K boundary for single-line resolution. If you position the P/M memory incorrectly, the P/M data will not be displayed. Since the Atari will be equipped with a varying amount of memory, it must be able to keep track of the amount available so that it knows where to locate the display data and display list. This is done at memory location 106 (RAMTOP). If you PEEK this location, you'll find the number of pages, *not the number of bytes*, in your machine. You can get the number of bytes by multiplying by 256.

POKEing into this location can be very useful for the programmer. One example is the location of large machine language programs that must be placed in a secure location that is not touched by BASIC. One way to accomplish this is to POKE a lower number of pages into RAMTOP, fooling the computer into believing that it has less memory than is the case. Then you can load your machine code in this safe hiding place, yet still access it when needed.

Another use is as a safe location for a redefined Atari character set. Again, there is one restriction. The relocated display data cannot cross a 4K boundary (graphics modes up to 7). If you don't observe this restriction, you'll find that you will be unable to plot and draw on part of the screen. RAMTOP for GRAPHICS 8 must be lowered in multiple 4K blocks. If you try it otherwise, you'll see weird and unwanted displays on your screen.

Thus, two methods for storing P/M data are:

1. Beneath the display list
2. Above a lowered RAMTOP

An example calculation for method one is shown below. However, it is tedious to have to calculate each time you program, so I've provided Table 3 for your use. This table gives the correct number of pages to offset the P/M system for either method of storage.

I hope these tables aid you in using the P/M and graphics systems. The systems are powerful, and their use will result in increasingly sophisticated displays.

Table 2. Example of P/M Positioning in Memory

Assume you wish to run P/M graphics mode 7. You want to use all four players, so all of the player/missile memory must be free and clear of the Screen Display memory.

	Required Step-Back In Memory, Pages
GRAPHICS 7, Screen Display (Table 1)	17
P/M graphics, single-line resolution (requires 2K)	8
Total =	25 pages

However, 25 pages is not on a 2K boundary:

6K = 24 pages
8K = 32 pages

Therefore, you must step back 32 pages for proper positioning of the P/M system.

Table 3. Player/Missile Positioning

Graphics Mode	Above RAMTOP* Double Resolution	Above RAMTOP* Single Resolution	Beneath Display List† Double Resolution	Beneath Display List† Single Resolution
0	4	8	8	16
1	4	8	8	16
2	4	8	8	16
3	4	8	8	16
4	4	8	8	16
5	4	8	12	16
6	4	16	16	24
7	12	16	24	32
8-11	16	16	36	40

*Number of pages to lower RAMTOP. Locate PMBASE at new RAMTOP.
†Locate PMBASE at indicated offset (in pages) below RAMTOP.

Notes

1. RAMTOP (location 106) defines the top of available memory. The display data lies just beneath RAMTOP. The display list resides just beneath the display data.
2. When lowering RAMTOP, the display data memory area must not cross a 4K boundary. RAMTOP for GR.8 must always be lowered in 4K increments.
3. Player/missile offsets are calculated by observing the following restrictions for the location of PMBASE:

	Double Resolution	Single Resolution
Offset from any other data	1K	2K
Boundary location for PMBASE	1K	2K

7

Memory Protection

Jim Clark

Player/missile graphics, redefined character sets, screen flipping, and machine language subroutines all create a need for protected areas of memory. This article covers several ways of handling the problem and gives you a program that protects low memory.

On the Atari, a problem arises in applications requiring a portion of memory to be protected from BASIC. For example, most machine language subroutines need protection. The problem is that BASIC is likely to use memory anywhere within available RAM, thus writing over the machine language subroutine and destroying it.

In many computers it is possible to protect memory at the "high" end, that is, at the highest RAM address. The Atari uses high memory for the data which is displayed on the screen. If you attempt to protect memory above the screen display by reducing the high memory value that BASIC thinks it has, then you cannot clear the screen or scroll text in any of the split-screen modes because these actions affect memory *beyond* the screen display area. These actions cause no problem when the screen display is actually the last thing in memory, because they apply to nonexistent memory. However, if you want to use memory beyond the screen display for your own purposes, then your data will be damaged by any action in your program which clears the screen or scrolls text in a text window.

Another alternative is to protect low memory. The main problem with this approach is that the memory protection must be done before BASIC gets control, since BASIC starts saving any program you enter beginning at the low memory address. The program shown here solves this problem as follows: it takes control of the Atari with a machine language subroutine and resets the system's low memory pointer. It then reinitializes BASIC—just as if you had pressed the SYSTEM RESET key—and BASIC takes control again, blissfully unaware that it now has less RAM to work with than it did before you ran this program.

To find the address of the memory you have protected, type

?PEEK(743) + 256*PEEK(744) *before* you run this program. The number printed can be used as the origin for a machine language subroutine, or as the destination address for whatever data you want to store in the protected area.

When you run the program, it asks how much memory you want to protect. Type in any positive number which is less than the amount of RAM you have available, as determined by typing ?FRE(0). The program reinitializes BASIC, and if you type ?PEEK(743) + 256*PEEK(744) again, the number printed will be greater than the value shown before running the program: the difference is the amount you requested to be protected. The memory area will remain protected until you turn the computer off, and the area can be used for machine language subroutines, redefined character sets, player/missile graphics objects, or any other use you might wish.

Memory Protection

```
90 REM MEMORY PROTECTION
100 REM ** LOAD MACHINE LANGUAGE SUBR
    OUTINE **
110 PGMSIZ=24:DIM SUBR$(PGMSIZ)
120 FOR I=1 TO PGMSIZ
130 READ BYTE
140 SUBR$(I)=CHR$(BYTE)
150 NEXT I
200 REM ** GET AMOUNT OF MEMORY TO PR
    OTECT **
210 ? "How many bytes do you want to
    protect";
220 INPUT PROTECT
230 HI=INT(PROTECT/256):LOW=PROTECT-2
    56*HI
240 SUBR$(6,6)=CHR$(LOW)
250 SUBR$(14,14)=CHR$(HI)
300 REM ** REINITIALIZE BASIC WITH TH
    E NEW LOW MEMORY POINTER **
310 Z=USR(ADR(SUBR$))
400 REM ** MACHINE LANGUAGE SUBROUTIN
    E **
410 REM MEMLO =$02E7;BOTTOM OF AVAILA
    BLE USER MEMORY
420 REM WARMST=$08;WARM START FLAG
430 REM CARTA =$A000;BASIC CARTRIDGE
    ENTRY POINT
440 REM
```

```
450 REM THE PROGRAM IS COMPLETELY REL
    OCATABLE, SO NO STARTING ADDRESS
    IS PROVIDED
470 REM
500 REM CLC ;INITIALIZE FOR ADDITION
510 DATA 24
520 REM LDA MEMLO ;ADD LEAST-SIGNIFIC
    ANT BYTES
530 DATA 173,231,2
540 REM ADC #PROTECT&$FF
550 DATA 105,0
560 REM STA MEMLO
570 DATA 141,231,2
580 REM LDA MEMLO+1 ;ADD MOST-SIGNIFI
    CANT BYTES
590 DATA 173,232,2
600 REM ADC #PROTECT/256
610 DATA 105,0
620 REM STA MEMLO+1
630 DATA 141,232,2
640 REM LDA #0 ;RESET THE WARM START
    FLAG
650 DATA 169,0
660 REM STA WARMST
670 DATA 133,8
680 REM JUMP CARTA ;START BASIC OVER
    AGAIN
690 DATA 76,0,160
999 END
```

7

Beware the RAMTOP Dragon

K. W. Harms

While the previous article showed how to protect low memory, this one tackles the problem of shielding high memory.

You've just had a brilliant idea for a program which requires some protected memory. Perhaps a special display list or character set is needed, or maybe a direct access memory "file." This article explains how to set aside that memory so that nothing will fiddle with it. Further, we'll reveal the generally unknown habits of the RAMTOP Dragon and show you three ways to make sure he doesn't gobble up your data.

The Atari offers a simple way to control how memory is internally managed by the operating system. In the previous article, Jim Clark shows how to move the lower boundary. Both Clark's method and the one discussed here protect memory from BASIC programs.

The map gives a very simple picture of Atari's memory management. Fixed memory boundaries are presented in decimal "addresses," but boundaries which vary according to the amount of memory in your machine or the program loaded at a particular time are given names such as "RAMTOP." The 400 and 800 both use the same system.

When you turn on the Atari with a BASIC cartridge, it takes a few seconds before "READY" to check out the machine and enter values for the boundaries into specific locations. PEEK allows you to look at those values. For instance, the value for RAMTOP is stored in address 106. The instruction"? PEEK(106)" will tell you where the Atari thinks the end of RAM is. Appendix I of the *Atari BASIC Reference Manual* explains that the value in 106 is in "pages" of 256 bytes. Multiplying number of pages times 256 gives the last address BASIC thinks it can use (for example, a PEEK(106) of "32" equals an address of 8192 or "8K").

The 400/800 always places the display list and display data immediately below RAMTOP. If you alter the value in RAMTOP, the Atari will push the display list and display data "downward" in RAM. This reduces the space available for your program, but leaves free RAM above the new (fake) RAMTOP. Since the Atari doesn't know about this space, it doesn't use it, usually. This is the space usually considered "reserved" for you.

Program 1 shows how to lower RAMTOP by four pages (1024 bytes). Remember to issue a GRAPHICS command immediately after moving RAMTOP so that the display list and data are moved below the new RAMTOP. Since line 60 will clear the screen, write down the old amount of free RAM and RAMTOP. Comparing them to the new numbers from lines 80 and 90 will show that RAMTOP is now lower and that less space is available for programs. That extra memory is now above RAMTOP and "reserved" for your exclusive use. RAMTOP is reset only by the RESET switch (and powering down/up), so that successively RUNning Program 1 will keep lowering RAMTOP until you run out of memory.

Although others have described ways to use the reserved space, they have not warned you about the RAMTOP Dragon who will periodically visit your reserved RAM and gobble up memory. Extensive field observations have revealed that the Dragon visits upper memory on three occasions:

1. A GRAPHICS command clears the visible screen and also the first 64 bytes above RAMTOP.
2. A CLEAR command (or "PRINT CHR$(125)") clears the first 64 bytes above RAMTOP.
3. Scrolling the *text window* of a graphics mode 3-8 screen clears up to 800 bytes above RAMTOP.

Program 2 lets you play with the RAMTOP Dragon. Lines 100 to 140 move down RAMTOP and reset the display list and data. Answer "NO" for all except the first pass, or the program will lower RAMTOP each time until you are out of memory.

After an initial questioning, the next section (lines 200 to 290) first turns off the "direct memory access" for the ANTIC processor so that the program will operate faster. It then fills the 900 bytes after RAMTOP with a sequence of values between 1 and 255. Note that the values will remain there as long as there's power to the CPU (and nothing clears them). Therefore, it's not necessary to repeat this step on subsequent RUNs.

The "Choose Action" section (lines 300 to 340) GOSUBs to the three major program sections.

The "Screen Play" routine (lines 1000 to 1100) exercises all three of the actions which call the Dragon. It clears the screen, changes graphics modes, and scrolls text windows. To scroll a window, enter graphics mode 3 to 8, and then enter numerical responses for as long as you wish to scroll (the amount of scrolling appears to affect the amount of memory cleared).

The "Check Memory" routine (lines 2000 to 2100) prints addresses for the first and last positions of reserved memory and requests the starting and ending addresses you wish to check. This section allows you to look at different ranges of locations to see how much memory has been cleared by displaying these memory addresses and their values. Knowing that the Dragon always leaves 0's in his path, and remembering that we loaded memory with values between 1 and 255, 0's will appear only in areas he visited. (Actually, I'm not sure whether the Dragon is a he or a she.) When you're done checking and want to enter a different set of actions, a "0,0" entry will return you to the "Choose Action" section.

The "Neat Trick of the Week" is found in lines 2055 and 2075. The memory address at 53775 can be used to tell you whether a key on the keyboard is being pressed at the time you PEEK it. If a key (any key) is depressed, 53775 contains the value 251. When the key is released, 53775 will show a 255. Line 2055 then stops the program whenever any key is pressed and restarts it when the key is released. Then POKEing 764 with a 255 (line 2075) clears the "halt" character so that future INPUTs, GETs, etc., aren't confused.

How can one avoid the Dragon anyway? There are many ways. You could never change graphics modes or clear or scroll the screen. This is difficult if you have any significant screen output. However, since the screen clear erases only 64 bytes, you could always clear the screen before the text window scrolls and never use those first 64 bytes. Or you could skip the first 800 bytes after RAMTOP and allow both scrolls and clears. Taking the other path, you could move the bottom of memory up and use memory below the new bottom (review Clark's article). However, this requires using a (simple) machine language subroutine.

If you are using the reserved memory in a *stable* program (one with no further coding), you have another choice. Program 3 shows how to use memory below RAMTOP as your special area instead of reserving memory above RAMTOP. In your program, wait until after *all* strings and arrays are dimensioned. Then, go to the highest resolution graphics mode and PEEK at the top of your BASIC program (line 10000). Since the Atari saves data on GOSUB and FOR/NEXT statements as it encounters them in a dynamic "stack" at the top of a program, you must provide some room for this storage. Figure on four bytes for each *active* GOSUB (one which hasn't been RETURNed), plus 16 bytes for each *active* FOR/NEXT (while it's FORing and NEXTing). Add this allowance to the previous address (line 10010) and use the total as the *bottom* of your reserved area.

Next, PEEK at MEMTOP, the top of RAM available for BASIC programs (line 10100), and use that number as the *top* of your area.

This method gives you the greatest possible amount of RAM without special code, but brings three general risks. If your BASIC program grows (by encountering an unexpected DIM or FOR/NEXT, for instance) after you have set the lower boundary, it will gnaw into the bottom of "your" memory. If the graphics mode is changed to a higher resolution mode after the upper boundary is set, the display list will push down into the reserved memory. Last, a program loaded after the boundaries are set may be larger and run into the set-aside memory.

The next time you see the RAMTOP Dragon, you'll be ready!

Program 1. Lower RAMTOP

```
10 ? "FREE RAM = ";FRE(0)
20 RAMTOP=PEEK(106):? "RAMTOP = ";RAM
   TOP;" PAGES":? "LAST ADDRESS = ";R
   AMTOP*256
30 FOR W=1 TO 1000:NEXT W
40 SMALLRAM=RAMTOP-4
50 POKE 106,SMALLRAM
60 GRAPHICS 0
70 RAMTOP=PEEK(106)
80 ? "NEW FREE RAM = ";FRE(0)
90 ? "NEW RAMTOP = ";RAMTOP;" PAGES":
   ? "LAST ADDRESS = ";RAMTOP*256
100 ? "RESERVED MEMORY BEGINS AT ";RA
    MTOP*256+1
```

Memory Management

57344-65535	ROM for character set, OS, etc.
55296-57343	ROM, Floating Point ROM
53248-55295	ROM, Hardware Registers ROM
	Unused space
40960-49151	CARTRIDGE ROM for BASIC, etc.
RAMTOP	END of RAM PEEK(106)
	Display List and Display Data (Usually 1K in Graphics 0, around 8K in Graphics 8)
MEMTOP	Top of RAM usable by BASIC programs PEEK(741) + 256*PEEK(742)
PROTOP	Program top for the current BASIC program PEEK(14) + 256*PEEK(15)
	Free RAM used for programs, data storage, etc.
BASIC LOMEM	Start of BASIC program
	Operating System, various buffers, hardware registers, etc.
0	Start of RAM addresses

Program 2. Move RAMTOP

```
50 REM STEP UP VARIABLES FOR CALLS
60 CHECK=1000:SCREEN=2000:QUIT=3000:D
   IM AN$(10):? CHR$(125)
100 REM MOVE DOWN RAMTOP
110 RAMTOP=PEEK(106)
120 ? "MOVE DOWN RAMTOP";:INPUT AN$:I
    F AN$(1,1)="N" THEN 200
130 RAMTOP=RAMTOP-5:POKE 106,RAMTOP
140 GRAPHICS 0
200 REM FILL 900 BYTES ABOVE RAMTOP
210 FIRST=RAMTOP*256+1:LAST=RAMTOP*25
    6+900
220 ? "FILL MEMORY ABOVE RAMTOP";:INP
    UT AN$:IF AN$(1,1)="N" THEN 300
230 POKE 559,0:REM TURN OFF SCREEN RE
    FRESHER
240 FOR POSITION=FIRST TO LAST
```

```
250 IF VALUE=255 THEN VALUE=0
260 VALUE=VALUE+1
270 POKE POSITION,VALUE
280 NEXT POSITION
290 POKE 559,34:REM TURN ON SCREEN
300 REM CHOOSE ACTION
310 ? :? "WHAT ACTION?":? "1 TO CHECK
    RAM":? "2 TO PLAY WITH SCREEN":?
    "3 TO QUIT"
320 INPUT ACTION
330 ON ACTION GOSUB SCREEN,CHECK,QUIT
340 GOTO 300
1000 REM SCREEN PLAY
1010 ? "CLEAR SCREEN":INPUT AN$
1020 IF AN$(1,1)="Y" THEN ? CHR$(125)
1030 ? "CHANGE GRAPHICS MODE";:INPUT
     AN$
1040 IF AN$(1,1)="Y" THEN ? "WHAT MOD
     E";:INPUT MODE:GRAPHICS MODE
1050 IF MODE<>0 THEN ? "ENTER ANSWERS
      UNTIL DONE, THEN NO";:INPUT AN$
1060 IF AN$(1,1)<>"N" THEN GOTO 1050
1070 IF MODE<>0 THEN GRAPHICS 0
1100 RETURN
2000 REM CHECK MEMORY
2010 ? :? "FIRST POSITION = ";FIRST:?
      "LAST = ";LAST:? "ENTER POSITIO
     NS TO CHECK OR 0,0 TO  RETURN"
2020 INPUT START,FINISH:IF START=0 TH
     EN GOTO 2100
2030 POKE 82,7:POKE 201,11:? :REM MOV
     E MARGIN, SET TAB
2040 FOR POSITION=START TO FINISH
2050 VALUE=PEEK(POSITION):? POSITION;
     " = ";VALUE,
2055 HALT=PEEK(53775):IF HALT=251 THE
     N GOTO 2055
2060 NEXT POSITION
2070 POKE 82,2:REM RESTORE MARGIN
2075 POKE 764,255
2080 GOTO 2000
2100 RETURN
3000 REM QUIT
3010 ? "NORMAL END OF JOB":END
```

Program 3. Using Memory Below RAMTOP

```
10000 PROTOP=PEEK(14)+256*PEEK(15)
10010 MEMSTART=PROTOP+24+1:REM START
      OF YOUR MEMORY; ALLOWS FOR 2 GO
      SUBS AND 1 FOR/NEXT
10100 MEMTOP=PEEK(741)+256*PEEK(742)
10110 MEMFINISH=MEMTOP:REM END OF YOU
      R MEMORY AREA
```

Listing Conventions

In order to make special characters, inverse video, and cursor characters easy to type in, *COMPUTE!* Magazine's 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.

Atari Conventions

Characters in inverse video will appear like: INVERSE VIDEO
Enter these characters with the Atari logo key, {⋀}.

When you see	Type	See	
{CLEAR}	ESC SHIFT <	↖	Clear Screen
{UP}	ESC CTRL -	↑	Cursor Up
{DOWN}	ESC CTRL =	↓	Cursor Down
{LEFT}	ESC CTRL +	←	Cursor Left
{RIGHT}	ESC CTRL *	→	Cursor Right
{BACK S}	ESC DELETE	◂	Backspace
{DELETE}	ESC CTRL DELETE	◧	Delete Character
{INSERT}	ESC CTRL INSERT	◨	Insert Character
{DEL LINE}	ESC SHIFT DELETE	◩	Delete Line
{INS LINE}	ESC SHIFT INSERT	◪	Insert Line
{TAB}	ESC TAB	▸	TAB key
{CLR TAB}	ESC CTRL TAB	▣	Clear TAB
{SET TAB}	ESC SHIFT TAB	▣	Set TAB stop
{BELL}	ESC CTRL 2	◳	Ring Buzzer
{ESC}	ESC ESC	↰	ESCape 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 ▣} means to enter five inverse-video CTRL-U's.

Index

Notes

Notes

COMPUTE!'s SECOND BOOK OF ATARI GRAPHICS

Like *COMPUTE!'s First Book of Atari Graphics*, this book contains many articles and tools that Atari programmers will find extremely useful. And some full-fledged graphics utilities that not only teach programming, but are also as exciting to use as commercial computer art software.

The graphic utilities "Screenbyter" and "Fontbyter" are designed to allow you to create, save, and add intricate graphic displays to your programs in a fraction of the time it would otherwise take.

But the book doesn't stop there. For instance, articles are included that: teach the beginners a simple way to add color to their programs, show the intermediate programmer how to use player/missile graphics, and help the advanced programmer protect memory from a BASIC program, as well as many more, useful programs and articles.

ISBN 0-942386-28-0

$12.95