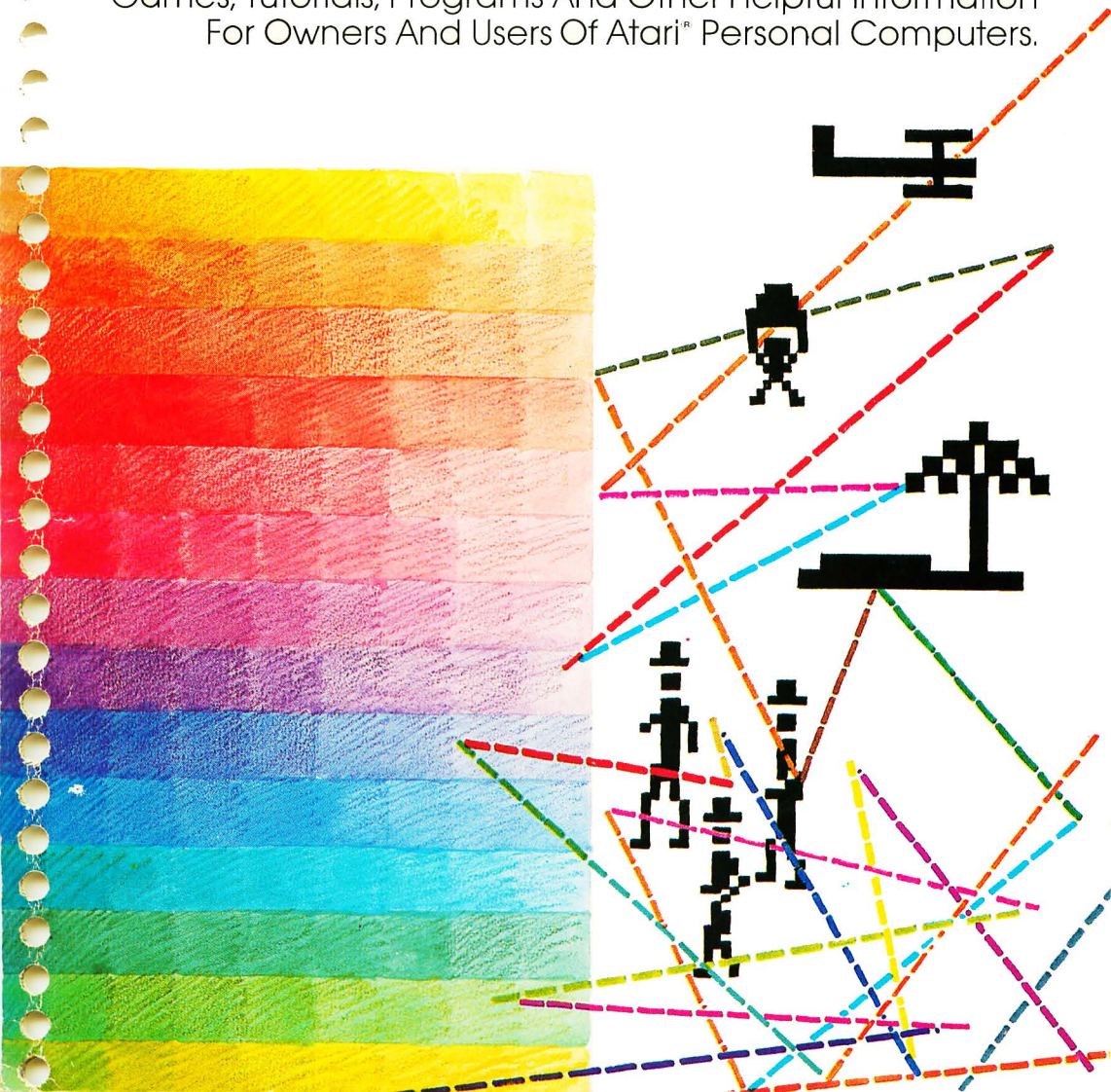
A **COMPUTE! Books** Publication

$12.95 US

COMPUTE!'S FIRST BOOK OF

ATARI® GRAPHICS

Games, Tutorials, Programs And Other Helpful Information For Owners And Users Of Atari® Personal Computers.



From The Editors Of **COMPUTE!** Magazine

COMPUTE!'S FIRST BOOK OF ATARI® GRAPHICS

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

ATARI is a registered trademark of Atari, Inc.

A
Small System
Services, Inc.
Publication

Copyright © 1982, Small System Services, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

"Using Strings For Graphics Storage" and "Copy Your Screen To Your Printer" were originally published in **COMPUTE!** Magazine, May 1981, copyright 1981, Small System Services, Inc. "Using The COLOR And LOCATE Instructions To Program Pong-Type Games" and "Positioning P/M And Regular Graphics In Memory" were originally published in **COMPUTE!** Magazine, September 1981, copyright 1981, Small System Services, Inc. "Discovering 'Hidden' Graphics" and "Adding High-Speed Vertical Positioning To P/M Graphics" were originally published in **COMPUTE!** Magazine, December 1981, copyright 1981, Small System Services, Inc. "Put Graphics Modes 1 And 2 At The Bottom Of Your Screen" and "P/M Graphics Made Easy" were originally published in **COMPUTE!** Magazine, February 1982, copyright 1982, Small System Services, Inc. "Printing Characters In Mixed Graphics Modes" was originally published in **COMPUTE!** Magazine, April 1981, copyright 1981, Small System Services, Inc. "Add A Text Window To Graphics 0," "A Self-Modifying P/M Graphics Utility," and "GRAPHICS 8 In Four Colors Using Artifacts" were originally published in **COMPUTE!** Magazine, June 1982, copyright 1982, Small System Services, Inc. "Mixing Graphics Modes 0 and 8" was originally published in **COMPUTE!** Magazine, June 1981, copyright 1981, Small System Services, Inc. "Character Generation" was originally published in **COMPUTE!** Magazine, February 1981, copyright 1981, Small System Services, Inc. "Designing Your Own Character Sets" was originally published in **COMPUTE!** Magazine, March 1981, copyright 1981, Small System Services, Inc. "SuperFont" was originally published in **COMPUTE!** Magazine, January 1982, copyright 1981, Small System Services, Inc. "TextPlot" was originally published in **COMPUTE!** Magazine, November 1981, copyright 1981, Small System Services, Inc. "Using TextPlot For Animated Games" was originally published in **COMPUTE!** Magazine, April 1982, copyright 1982, Small System Services, Inc. "Animation And P/M Graphics" and "Atari Video Graphics And The New GTIA, Part II" were originally published in **COMPUTE!** Magazine, August 1982, copyright 1982, Small System Services, Inc. "Extending Player/Missile Graphics" and "Beware The RAMTOP Dragon" were originally published in **COMPUTE!** Magazine, October 1981, copyright 1981, Small System Services, Inc. "Extra Colors Through Artifacting" was originally published in **COMPUTE!** Magazine, May 1982, copyright 1982, Small System Services, Inc. "Atari Video Graphics And The New GTIA, Part I" was originally published in **COMPUTE!** Magazine, July 1982, copyright 1982, Small System Services, Inc. "Atari Video Graphics And The New GTIA, Part III" was originally published in **COMPUTE!** Magazine, September 1982, copyright 1982, Small System Services, Inc. "Memory Protection" was originally published in **COMPUTE!** Magazine, July 1981, copyright 1981, Small System Services, Inc. "Screen Save Routine" was originally published in **COMPUTE!** Magazine, March 1982, copyright 1982, Small System Services, Inc.

Printed in the United States of America

ISBN 0-942386-08-6

10 9 8 7 6 5 4 3 2 1

v. Introduction . . . . . Robert C. Lock

1 Chapter One: Fundamentals Of Atari Graphics
3 The Basics Of Atari Graphics . . . . . Tom R. Halfhill
16 Using Strings For Graphics Storage . . . . . Michael Boom
20 Using The COLOR And LOCATE Instructions To Program Pong-Type Games . . . . . Michael A. Greenspan

23 Chapter Two: Customizing The Graphics Modes
25 How To Design Custom Graphics Modes . . . . . Craig Chamberlain
37 Put Graphics Modes 1 And 2 At The Bottom Of Your Screen . . . . . R. Alan Belke
41 Printing Characters In Mixed Graphics Modes . . . . . Craig Patchett
44 Add A Text Window To GRAPHICS 0 . . . . . Charles Brannon
46 Mixing Graphics Modes 0 And 8 . . . . . Douglas Crockford

51 Chapter Three: Redefining Character Sets
53 Designing Your Own Character Sets . . . . . Craig Patchett
62 SuperFont . . . . . Charles Brannon
77 Character Set Utilities . . . . . Fred Pinho

89 Chapter Four: Amination With Character Graphics
91 TextPlot . . . . . Charles Brannon
98 Using TextPlot For Animated Games . . . . . David Plotkin
108 High-Speed Animation With Character Graphics . . Charles Brannon

127 Chapter Five: Animation With Player/Missile Graphics
129 Introduction To Player/Missile Graphics . . . . . Bill Wilkinson
140 A Self-Modifying P/M Graphics Utility . . . . . Kenneth Grace, Jr.
154 Adding High-Speed Vertical Positioning To P/M Graphics . . . . . David H. Markley
164 P/M Graphics Made Easy . . . . . Tom Sak and Sid Meier
172 Animation And P/M Graphics . . . . . Tom Sak and Sid Meier
184 Extending Player/Missile Graphics . . . . . Eric Stoltman
188 The Collision Registers . . . . . Matt Giwer
192 The Priority Registers . . . . . Bill Wilkinson

201 Chapter Six: Advanced Graphics Techniques
203 GRAPHICS 8 In Four Colors Using Artifacts . . . . . David Diamond
208 Atari Video Graphics And The New GTIA, Part 1 . Craig Chamberlain
215 Atari Video Graphics And The New GTIA, Part 2 . Craig Chamberlain
224 Atari Video Graphics And The New GTIA, Part 3 . Craig Chamberlain
236 Protecting Memory For P/M And Character Sets . . . . . Fred Pinho
239 Screen Save Routine . . . . . Joseph Trem

245 Listing Conventions (Guide To Typing In Programs)
246 Index

Introduction

Robert Lock, Publisher/Editor-In-Chief, **COMPUTE!** Publications

This special addition to our First Book Series represents the first time we've published a theme-specific book. *COMPUTE!'s First Book of Atari Graphics* contains published as well as original, unpublished material that has been carefully chosen to provide any Atari user with helpful, useful information on the extensive capabilities available with Atari graphics.

As with our parent publication, **COMPUTE!** Magazine, you'll find a range of tutorials, programs, and more, for the beginner to the most advanced, ready to type right into your computer and use.

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

Special thanks to Tom R. Halfhill, our Features Editor, who bore the organizational brunt of this volume, and to the entire editorial and production staffs who assisted in this, our ninth book from **COMPUTE! Books**. Cover design: Georgia Papadopoulos. Cover illustration: Harry Blair.

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

Chapter 1

Fundamentals Of Atari Graphics

The Basics Of Atari Graphics

Tom R. Halfhill

If you are new to the Atari and have acquired a bit of familiarity with BASIC, but have not yet taken the plunge into graphics, this article will introduce you to the fundamentals.

For some reason, many people are intimidated by the programming steps required to create computer graphics. Probably this is because creating computer graphics is not as easy as it looks. The typical buyer of a personal computer is dazzled in the store by all the fantastic arcade games and impressive graphics demos with which the sales people are armed. It all looks so simple. Then the buyer eagerly unpacks the computer at home and quickly discovers that even crude pictures cannot be created without screenfuls of cryptic programming that seemingly have more in common with Sanskrit than English.

But there is hope. It's not really that hard – honest. Nobody is promising that you'll be able to duplicate *Star Raiders* or *Pac-Man* any time soon, but the basics of computer graphics are quite easy to grasp for anyone who has some knowledge of BASIC programming. You don't need to be a math wizard, either. The most valuable attributes are a willingness to learn and to experiment. And, of course, to be creative.

Choosing A Graphics Mode

Atari graphics are particularly challenging to learn, mainly because the Atari computers have extremely versatile graphics. Luckily, Atari made it easier for us by including many special keywords in Atari BASIC that are dedicated to graphics. The first step, then, is to learn those keywords. And by the way, if you don't already have your *Atari BASIC Reference Manual* handy, take a second to grab it. This book and the *Manual* should help to explain each other.

The most basic of the keywords is the GRAPHICS com-

mand. This tells the computer which graphics mode you want, which in turn determines how the screen will look. The format is GRAPHICS (aexp), where (aexp) is any arithmetic expression that results in a positive integer (in other words, not a negative number or a fraction). For example, GRAPHICS 6 is a valid command which tells the computer you want graphics mode six. GRAPHICS 3+3 or GRAPHICS 3*2 would do the same thing.

Depending upon how old your Atari is, the GRAPHICS command gives you access to either nine or twelve different graphics modes. The reason for the difference is that earlier Ataris (generally, those shipped before late 1981) came with a TV controller chip called the CTIA. Later Ataris have a GTIA chip instead. The chips are fully compatible – programs written on CTIA Ataris will run on GTIA machines and vice versa – but the GTIA adds three new graphics modes. Users with CTIA chips can have their computers upgraded if they wish. (See "Atari Video Graphics And The New GTIA" in Chapter 6.)

So, you have either nine or twelve basic graphics modes to choose from. In addition, most of them have two variations, for a total of up to 20 modes.

The modes are of two main types: pure graphics modes and text modes. The first three modes – GRAPHICS 0, 1, and 2 – are text modes. When you switch on an Atari with a BASIC cartridge plugged in, it defaults to GRAPHICS 0. GRAPHICS 0 has 24 horizontal rows of up to 40 characters each on the screen. (If you've counted only 38 characters, it's because the left margin is pre-adjusted to allow for TVs which *overscan*, or cut off the left edge of the screen image.) GRAPHICS 1 and 2 display larger-size characters. GRAPHICS 1 characters are the same height as those in GRAPHICS 0, but are twice as wide. GRAPHICS 2 characters are not only twice as wide, but also twice as tall.

The graphics modes generally used for creating pictures are GRAPHICS 3 through 8 (3 through 11 on GTIA machines). GRAPHICS 3 through 8 are *mixed modes*. That is, they are combinations of text and graphics modes. For example, type GRAPHICS 3 into the Atari. You'll see a black screen with a small blue rectangle at the bottom. That rectangle is called the *text window*. Although the upper part of the screen is a graphics mode for drawing pictures, the text window is a section of GRAPHICS 0 for displaying text. Think of it as the term implies:

a "wall" of GRAPHICS 3 with a "window" of GRAPHICS 0.

The GRAPHICS 0 text window appears in all the graphics modes from three through eight. Separate commands, which we'll soon learn, are required to display graphics or text in each part of those screens.

If you want a "pure" graphics mode – a full screen for graphics with no GRAPHICS 0 text window – simply add 16 to the mode number of the GRAPHICS statement. For example, GRAPHICS 3 + 16 switches the screen to GRAPHICS 3 without a text window. Some programmers would type GRAPHICS 19, which is the same thing. Adding 16 works for all the modes except GRAPHICS 0, which ordinarily cannot display a separate text window.

Just Like Graph Paper

You may be wondering why there are so many graphics modes, and how to choose among them. The modes differ in three main ways: resolution, number of colors available, and memory consumed.

First, resolution. Think of the graphics screen as a sheet of graph paper. Some graph paper is divided into very small squares; other graph paper has larger squares. If you had to draw a picture on graph paper only by coloring in the squares – not by sketching lines – the graph paper with the smaller squares obviously would allow you to create a more detailed picture. It would allow greater *resolution.*

This is exactly how a computer screen works. The screen is divided into tiny squares, and graphics are created by "filling in" those squares. These squares are sometimes called *pixels,* for "picture elements." In the highest resolution modes, the pixels are so small that they do not appear as squares at all, but as tiny dots.

The Atari graphics modes offer different resolutions. The higher the graphics mode number, the greater the resolution. So you can draw much more finely detailed pictures in GRAPHICS 8, for instance, than in GRAPHICS 3. In GRAPHICS 8, there are 320 horizontal pixels (or "graph paper squares") per row on the screen; GRAPHICS 3 has only 40. So GRAPHICS 8 has a *horizontal resolution* of 320 and GRAPHICS 3 has a horizontal resolution of 40.

When figuring the *vertical resolution,* don't forget about the text window. These four lines of GRAPHICS 0 at the bottom of

the screen take up room that could be used for drawing pictures; thus, it decreases the vertical resolution. Adding 16 to the graphics mode number regains that resolution. So GRAPHICS 3, for example, has a vertical resolution of 20 pixels; GRAPHICS 3+16 has 24 pixels.

Table 1 shows the resolutions of the graphics modes with and without the text window.

Another difference is color. GRAPHICS 2 (the double-height, double-width text mode) normally can display characters in five colors at a time. GRAPHICS 4 and 6 can display only two colors. These differences also are shown in Table 1.

The final main difference between the Atari graphics modes is the amount of Random Access Memory (RAM), or user-available memory, they consume. You may have guessed that the first two characteristics – resolution and number of colors – determine the third. The higher the resolution, and the more colors available, the more memory is required. We won't delve into the details, but it's enough to know that the computer must keep track of what it is displaying, so the more it displays, the more memory it needs.

You don't have to worry about allocating the memory yourself; the computer automatically seizes the memory it needs when a GRAPHICS statement is executed. But you do have to worry about how much memory you have left. A 16K RAM Atari, for example, normally has about 13,300 of its 16,000 memory bytes free when first switched on (the remainder is also allocated by the computer for other uses, but we won't go into that here). Entering GRAPHICS 8 instantly chops that down to about 5200 bytes, or 5.2K, because GRAPHICS 8 requires about 8000 bytes just to set itself up. That doesn't leave much room for an involved program. In fact, the original 8K Ataris cannot even enter GRAPHICS 8 without memory expansion.

Again, Table 1 shows how much memory each graphics mode consumes.

The Chameleon Computer

When we said before that the graphics modes are limited to displaying a certain number of colors, we didn't mean that you're stuck with the same colors all the time. Like a chameleon, the Atari can change its colors at will – your will.

How many colors can you choose from? If you have an

older CTIA chip in your machine, up to 128 colors are possible. With the new GTIA, there are 256.

These break down into 16 basic colors, with variable shades (or luminances) to achieve the 128 or 256 hues.

However, without resorting to the kind of special tricks described in the more advanced chapters of this book, a much smaller number of colors is available simultaneously.

All the graphics modes default to certain colors. It's easy to change these colors, though, with the SETCOLOR statement. The format is SETCOLOR (register), (hue), (luminance). These three values can be arithmetic expressions, but should evaluate to whole numbers. In addition, the values have certain ranges.

(Register) is a number from zero to four. The "registers" are really memory locations which control the screen colors. The foregrounds, backgrounds, and borders of the graphics modes are in turn controlled by these registers. For example, the backgrounds of GRAPHICS 1 through 7 are controlled by register four; since register four defaults to black, the backgrounds of those graphics modes appear on the screen as black.

(Hue) allows you to change that default color. You just plug in a color number from zero to 15 (remember, we said there were 16 basic colors). Table 2 shows the color numbers, and Table 3 the default colors for the registers.

(Luminance) simply adjusts the brightness, or shade, of the color selected by (hue). This must be an even number from zero to 14, with zero the darkest and 14 the brightest.

So, to change the background of GRAPHICS 3 from black to green, you could enter SETCOLOR 4,12,8.

That's it. You can change the color of any color register this way.

Drawing Pictures, At Last

We haven't forgotten that the whole reason you're reading this book is that you want to create graphics. But we had to get the basics out of the way first. Now for the nitty-gritty.

The graph paper analogy really comes in handy here. In fact, some actual graph paper often is an indispensable aid when you're planning complex drawings for a screen.

Picture the graphics screen again as a sheet of graph paper. Depending on the resolution of the graphics mode, the screen has certain coordinates. For instance, GRAPHICS 6 without

the text window (that is, GRAPHICS 6+16) has a horizontal resolution of 160 pixels and a vertical resolution of 96. Since computer work often involves counting from zero instead of one, the horizontal coordinates range from zero to 159, and the vertical coordinates from zero to 95. Lock the applicable coordinates in your head whenever working with a graphics mode, because if you exceed them, you'll encounter the dreaded ERROR- 141, CURSOR OUT OF RANGE.

Now, we said before that you didn't have to be a math wizard to program computers, and we meant it. In fact, plotting graphics coordinates is one case where a knowledge of higher math is actually a detriment. Mathematicians usually plot coordinates starting from the lower-left corner of a graph; computer designers start at the upper-left corner. So, according to the coordinate system we just described, position 0,0 is the upper-left corner of the TV screen in GRAPHICS 6, and all the graphics modes.

Look at the figure; it shows how the coordinates run in GRAPHICS 6+16. This is the same for all the modes, except that the upper limit of the coordinates will differ according to each mode's resolution. Coordinate position 159,95 is the lower-right corner in GRAPHICS 6+16; in GRAPHICS 5+16 it would be 79,39; and in GRAPHICS 8+16, 319,191. (The horizontal, or X, coordinate always precedes the vertical, or Y, coordinate.)

It's vital to understand how this coordinate system works; it is the basis for all drawing and positioning on the screen.

For example, to draw a dot on the screen, you "light up" or "switch on" the pixel at that location, according to its coordinates. This is done with the PLOT statement. The format is PLOT X,Y – where X is the horizontal coordinate and Y is the vertical coordinate. PLOT 0,0 will put a dot in the upper-left corner of the screen. The size of that dot depends on the graphics resolution; the higher the resolution, remember, the smaller the dot. PLOT 159,95 would draw a dot ("switch on a pixel") at the lower-right corner of the screen in GRAPHICS 6+16.

To draw a line, you could simply PLOT a number of dots in a row. For instance, PLOT 2,4:PLOT 2,5:PLOT 2,6 etc., would draw a short vertical line near the left edge of the screen. But there's an even easier way: the DRAWTO statement. The format is DRAWTO X,Y. DRAWTO does just what it implies; it draws a line to the horizontal and vertical coordinates specified. Before using DRAWTO, however, you have to include a PLOT

statement to give the DRAWTO a starting point. Afterward, DRAWTO will pick up where it left off. For instance, you could draw a square like this:

```
10 GRAPHICS 6+16:COLOR 1;PLOT 5,5:DRAWTO 10,5:
DRAWTO 10,10:DRAWTO 5,10:DRAWTO 5,5
```

Drawing In Different Colors

You probably noticed the COLOR statement in that last example and wondered where it came from. A COLOR command is necessary before executing any PLOTs or DRAWTOs. If you leave it out, the PLOTs and DRAWTOs will be displayed in the background color, rendering them invisible. The COLOR statement, then, selects the color for subsequent PLOT and DRAWTO statements. The format is COLOR (aexp), where (aexp) is any arithmetic expression that evaluates to a whole number (fractions are automatically rounded). Further, that number should be from zero to three.

Important: don't confuse COLOR with SETCOLOR. SETCOLOR selects the foreground, background, and border colors to be displayed by the color registers, while COLOR determines the color of points or characters to be plotted on the graphics screen. Since COLOR is the foreground (plotting) color, it can be changed with SETCOLOR.

A useful analogy is to think of the colors available on the Atari as a box of crayons (128 crayons with CTIA machines and 256 crayons with the GTIA chip). SETCOLOR allows you to select a handful of those crayons at once – the exact number depending on the graphics mode (see Table 1). In GRAPHICS 6 you can select two. Once you've chosen the crayons, COLOR allows you to choose which crayon the computer will use for subsequent PLOTs and DRAWTOs. At any time, you can execute COLOR to switch among the crayons in your hand, or SETCOLOR to replace the crayons in your hand with other colors from the box. But don't carry the analogy too far – when you change colors with SETCOLOR, everything you've already drawn changes color, too.

For example, in GRAPHICS 7, the color selected by the statement COLOR 1 is determined by the value in SETCOLOR register zero. The default color is orange. So if you PLOT and DRAWTO in GRAPHICS 7 with COLOR 1, the figure will appear orange. To get a green figure, you would execute SET-

COLOR 0,12,8:COLOR 1:PLOT, etc. The SETCOLOR statement would change color register zero from orange to green, and COLOR 1 would use the new color for all subsequent PLOTs and DRAWTOs.

Note that any previous figures plotted in orange would change to green instantly upon execution of the new SETCOLOR. This system is known as *color indirection* and accounts for the flashing screen colors you may have noticed in those fancy graphics demos you've admired. Yet, as you see, the technique is really very simple.

One thing that takes some getting used to is that the COLOR statement does not get its color from the same registers in all graphics modes, and some modes are restricted to only two colors. Refer to the table on page 53 of the *Atari BASIC Reference Manual* for a summary of how COLOR and SETCOLOR take effect in the various modes.

More Graph Paper

That graph paper analogy comes in handy again for two more graphics statements you'll need to learn.

The first is POSITION. The format is POSITION X,Y – where X is the horizontal coordinate of the graphics mode and Y is the vertical coordinate. POSITION is a lot like PLOT, except it doesn't draw anything. That is, POSITION X,Y directs the computer's attention to point X,Y on the screen just as PLOT X,Y does, except the pixel at that point is not "switched on." Instead, the invisible graphics cursor – similar to the text cursor you're familiar with in GRAPHICS 0 – is spotted at point X,Y in preparation for the next command.

This command could be a PRINT statement in one of the large text modes, GRAPHICS 1 or 2. For example, GRAPHICS 2:POSITION 5,5:PRINT #6;"HELLO" would print "HELLO" starting at column 5, row 5 on the GRAPHICS 2 graphics screen. ("PRINT #6;" merely specifies a PRINT to the graphics part of the screen; a PRINT statement without the "#6;" would print the message in the text window.) The POSITION statement is valuable for neatly formatting screens in your programs.

The LOCATE statement is another handy programming tool. The format is LOCATE X,Y,Z – again, where X and Y are the horizontal and vertical screen coordinates. The third variable, shown here as "Z," returns a value read from the pixel at point X,Y. That value depends on the graphics mode. In modes

three through eight, the value is the color register in use (the SETCOLOR number) at that pixel position. In GRAPHICS 1 and 2, the large text modes, the value tells which character as well as which color register is in use at the pixel position. And in GRAPHICS 0, the value is the ATASCII code for the character at that location (ATASCII is the character code system; see Appendix C of the *Atari BASIC Reference Manual).*

Since LOCATE can determine what is being displayed at a certain location on the screen, it is sometimes used to detect collisions (or impending collisions) between objects in games. (See "Using The COLOR And LOCATE Instructions To Program Pong-Type Games," later in this book.)

Beginning Animation

At this point, if you've been practicing and experimenting with the principles we've covered so far, you know all the basics you need to draw figures and colorful designs on the graphics screens. But you're probably wondering how to animate those images.

Animation is perhaps the most difficult graphics technique to master. For one thing, fast, smooth animation requires a great deal of processing speed, sometimes more than is possible with a relatively slow language such as BASIC. But it is possible, and there are several methods. We won't cover any of them in depth here, but we will introduce you to the simplest forms to whet your taste a bit.

One method may already have occurred to you. By just drawing a figure on the screen, erasing it, and re-drawing it at a slightly different location, you can achieve the illusion of movement in the same way that cartoonists do. You already know how to draw a picture with PLOT and DRAWTO. Erasing it is just as easy – you simply re-draw the image in the background color, making it disappear. Then you switch back to the foreground color, re-draw the figure elsewhere, and presto – it will seem to have moved. Sometimes this is called *playfield graphics.* Try Program 1 for an example.

Similarly, the POSITION and PRINT statements may have suggested another simple method of animation. Consider the text modes, GRAPHICS 0, 1, and 2. While commonly used for "title screens" and other applications requiring text displays, they also come in handy for a technique called *character graphics.* To make the character "A" seem to move across the screen, for

example, you PRINT it at the desired starting location, erase it by PRINTing a blank space in the same spot, and then re-PRINT it at the next location. POSITION lets you specify where the movement will start, and LOCATE can detect collisions with other characters.

All fine and good, you say, but why would I want to animate letters of the alphabet?

Have you ever noticed what happens when you hold down the CONTROL key and press an alphabetic key on the Atari? The resulting character is an odd shape of some sort. A number of these shapes are available, known as control characters. When PRINTed side-by-side, they can be put together to form robots, spaceships, or what-have-you. The POSITION and PRINT statements can supply the animation. This is sometimes called *control graphics.* Try Program 2 for an example.

Both of these methods – playfield graphics and control graphics – are straightforward and simple. Many fine games have been written in BASIC using these techniques. In fact, some computers have no other methods available. However, fast movement of complex figures does tend to get messy. Luckily, the Atari computers offer several more advanced techniques, such as redefined characters graphics (which allows you to sculpt that "A" into almost any shape you want), player/missile graphics, page-flipping, and screen scrolling.

Those techniques are covered later in this book. This article was merely intended to arm newcomers to Atari graphics with the basic tools needed to understand the more esoteric subjects. When you run into roadblocks – and you'll encounter them as you forge ahead into the sometimes tricky world of computer graphics – just keep your manuals handy and remember this famous American proverb:

"When all else fails, read the instructions."

Figure . Coordinates of GRAPHICS 6 + 16.

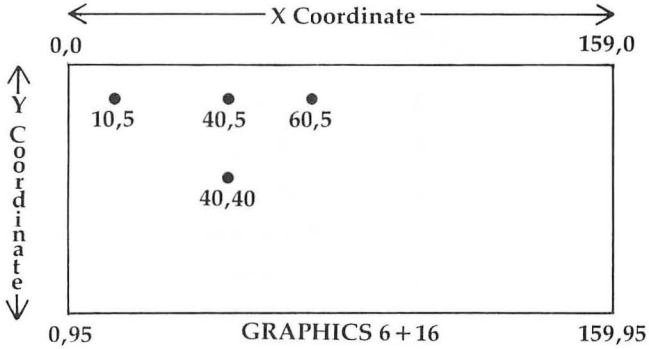


Table 1. Summary Of The Graphics Modes.

Graphics Mode	Resolution With Text Window	Resolution Without Text Window	Colors Available	Memory Consumed
0	—	40 x 24	2	993
1	20 x 20	20 x 24	5	513
2	20 x 10	20 x 12	5	261
3	40 x 20	40 x 24	4	273
4	80 x 40	80 x 48	2	537
5	80 x 40	80 x 48	4	1017
6	160 x 80	160 x 96	2	2025
7	160 x 80	160 x 96	4	3945
8	320 x 160	320 x 192	2	7900
9	—	80 x 192	16	7900
10	—	80 x 192	9	7900
11	—	80 x 192	16	7900

Table 2. Atari Color Numbers.

**Note:* Color TVs may vary.

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

Table 3. Color Register Default Values.

**Note:* Color TVs may vary.

Register Number	Color Number	Luminance Number	Color*
0	2	8	Orange
1	12	10	Green
2	9	4	Blue
3	4	6	Pink
4	0	0	Black

Program 1.

```
10 GRAPHICS 6:CHANGE=1:A=5:B=10:? "
   GRAPHICS 6:  PLAYFIELD ANIMATION"
20 FOR MOVE=1 TO 2
30 COLOR CHANGE:PLOT A,A:DRAWTO B,A:D
   RAWTO B,B:DRAWTO A,B:DRAWTO A,A
40 IF CHANGE=1 THEN CHANGE=0:NEXT MOV
   E
50 IF CHANGE=0 THEN CHANGE=1:NEXT MOV
   E
60 A=A+1:B=B+1
70 IF A>79 OR B>79 THEN GRAPHICS 2:PO
   SITION 0,6:? #6;"PLAYFIELD ANIMATI
   ON":? "     * GRAPHICS 2 TEXT WINDO
   W *":END
80 GOTO 20
```

Program 2.

```
10 GRAPHICS 0:A=0:B=10:DIM CHARACTER$
   (1):CHARACTER$="A"
20 POSITION A,B:? CHARACTER$
25 FOR SLOMO=1 TO 10:NEXT SLOMO
30 POSITION A,B:? " "
40 A=A+1:IF A>39 THEN 60
50 GOTO 20
60 IF CHARACTER$<>"{T}" THEN CHARACTE
   R$="{T}":A=0:B=10:GOTO 20
70 POSITION 10,5:? "CHARACTER ANIMATI
   ON":? "  WITH A LETTER AND CONTROL
    CHARACTER"
```

Using Strings For Graphics Storage

Michael Boom

If you've ever been frustrated attempting to PLOT and DRAWTO your way through a complex pattern or design in Atari graphics, you might appreciate a method of graphics generation using text strings to store pixel data. While this string method is not simpler to use in all cases, its ease of data entry and manipulation possibilities make it a strong graphics tool.

Simple line drawings over large areas of the screen are best done using PLOT and DRAWTO commands, since this method uses less memory and generates images faster than the string method will. However, if you have a very complex pattern in a small area of the screen, the string method works well. The heart of string graphics lies in the fact that if you run a PRINT #6 statement followed by ASCII characters while in graphics modes 3-7, colored pixels will appear on the screen. Different letters and symbols will plot different colors, but for our purpose we will deal only with the letters A, B, C, and D. Each of these letters plots a different colored pixel in graphics modes 3, 5, and 7:

A plots color 1 (color register #0)
B plots color 2 (color register #1)
C plots color 3 (color register #2)
D plots color 0 (color register #4)

In graphics modes 4 and 6, only the letters A and B need be used, A for the plotting color, B for the background color.

For a demonstration, typing the command

```
GRAPHICS 3: PRINT #6; "ABCDA"
```

moves the pixel string down and to the right.

Creating A Graphics String

We can now use the above methods to plot a pattern. First graph out the area needed for the pattern, then fill in the pattern using

"A", "B", "C", and "D" to represent the colors wanted:

```
String 1  CDDDDAAAAA
String 2  DCDDDDDDAA
String 3  DDCDDDDADA
String 4  DDDCDDADDA
String 5  DDDDCADDDA
String 6  AAAAACDDDD
String 7  ABBBADCDDD
String 8  ABCBADDCDD
String 9  ABBBADDDCD
String 10 AAAAACCCCC
```

Now break down the graph as a series of strings, in this case ten string of ten characters each:

String 1 is "CDDDDAAAAA"
String 2 is "DCDDDDDDAA"
etc.

Concatenate the ten strings for more efficient data storage:

```
"CDDDDAAAAADCDDDDDDAADDCDDDDADADDDCDDA
 DDADDDDCADDDAAAAAACDDDDABBBADCDDDABCBA
 DDCDDABBBADDDCDAAAAACCCCC"
```

We have now generated all the data necessary to plot our figure (a square with an arrow) in the graphics mode, and have stored it in one long string.

Display

To plot the string on the screen, determine where you would like the upper left-hand corner of the figure to be located, and enter it during the run of the following program after prompt "X,Y?".

```
 10 GRAPHICS 5
 20 DIM A$(100)
 30 A$="CDDDDAAAAADCDDDDDDAADDCDDDDAD
 ADDDCDDADDADDDDCADDDAAAAAACDDDDABBBAD
 CDDDABCBADDCDDABBBADDDCDAAAAACCCCC"
 40 PRINT "X,Y";:INPUT X,Y
 80 FOR K=1 TO 10
 90 POSITION X,Y+K-1
100 PRINT #6;A$(K*10-9,K*10)
110 NEXT K
```

In this program, lines 20 and 30 set up our main pixel data string, and line 40 establishes the upper left corner coordinates of the

figure. Lines 80 and 110 set up a loop of ten steps, to divide our main data string into seven rows. Line 90 positions the cursor for each row, and line 100 prints ten consecutive ten-character strings on the screen.

Obviously, there are figures which require strings too long for direct entry in Atari BASIC. In that case, divide the figure into several rectangular sections, each small enough for inclusion into one string (usually under 100 characters in length). Then concatenate the string as explained in the *Atari BASIC Reference Manual*, p. 39.

Figure Manipulation

Plotting a figure using string graphics is fairly simple and straightforward. Its real strength lies in figure manipulation through string reading. Some easy manipulations are:

1. Figure rotation (in 90° increments)
2. Figure inversion
3. Color changes

For figure rotation, using the same example figure and data string, let's substitute and add to the previous program. For a 90-degree turn clockwise, add and substitute:

```
20 DIM A$(100),B$(100
50 FOR K=1 TO 10: FOR L=1 TO 10
60 B$(K*10 - 10+L,K*10 - 10+L)=A$((10 - L)*10+K,(10 - L)
   * 10+K)
70 NEXT L, NEXT K
100 PRINT #6;B$(K*10 - 9,K*10)
```

For a 270-degree clockwise rotation, substitute to the above:

```
60 B$(K*10 - 10+L,K*10 - 10+1)+A$(L*10+1 - K,L*10+1 - K)
```

For a 180-degree clockwise rotation, substitute to the above:

```
50 FOR K=1 TO 100
60 B$(K,K)=A$(101 - K,101 - K)
70 NEXT K
```

To change color assignments, add and substitute to the *original* program:

```
50 FOR K=1 TO 100
60 IF A$(K,K)="C" THEN A$(K,K)="A"
70 NEXT K
```

To invert a figure, substitute to the original program:

```
100 PRINT #6;A$((11 - K)*10 - 9,(11 - K)*10)
```

To turn a figure left to right, substitute in the 180-degree rotation program:

```
100 PRINT #6;B$((11 - K*10 - 9,(11 - K)*10))
```

The string used to manipulate this 10 x 10 figure can easily be incorporated into subroutines for use in programs using repetitive figures in different positions. Further experimentation for more possibilities is definitely in order.

Using The COLOR And LOCATE Instructions To Program Pong-Type Games

Michael A. Greenspan

Here's the skeleton of a Pong-*type game that demonstrates simple Atari playfield graphics. When you grasp the principles, it will be easy to flesh out the program yourself.*

New Atari owners may be confused (as I was) about the COLOR and SETCOLOR instructions. These two commands, and the LOCATE instruction, form the basis of the following *Pong*-type game.

In GRAPHICS 3, there are four color registers labeled 0, 1, 2, and 3, which are accessed by the instruction COLOR X, where X is the number of the register desired. (COLOR 4 is the same as COLOR 0; COLOR 5 is the same as COLOR 1, etc.) While COLOR determines the register used, SETCOLOR enables you to determine which of the 128 colors are used by your chosen register to draw points on the screen. Thus, since the SETCOLOR instructions are identical, the following commands will each put a dark gold point on the screen at location 1,1:

```
10 GR.3: COLOR 1: SETCOLOR 0, 1, 2 : PLOT 1,1
10 GR.3: COLOR 2: SETCOLOR 0, 1, 2 : PLOT 1,1
```

**The SETCOLOR command instructs the computer to set the color of the points on the screen (that's the function of the 0) to color 1 (that's gold) brightness 2. A two for the first number will change the text window to that color. A four will change the background.*

Each color register has a different default color that determines the color of the points plotted in that register if no SET-

COLOR 0, X,X instruction is given. Therefore, plotting points in different color registers will produce different colors in the absence of SETCOLOR instructions, and identical colors if identical SETCOLOR instructions are used.

In the program below, a ball moves from left to right and a joystick maneuvers a paddle on the far right to intercept the ball. The paddle is plotted in color register 1, and the ball in color register 2. In order to move the ball, it is replotted in color register 4, whose default color is the same as the background color (and thus is invisible), and then replotted on the adjacent square in color register 2.

The LOCATE instruction determines if there is a hit. X and Y are the X and Y coordinates of the ball. LOCATE X+1, Y, X tells the computer to LOCATE the point to the right of the ball and to store the *color register* of that point in Z. Since the paddle is plotted in color register 1, Z=1 means that the ball hit the paddle.

Once you understand the use of COLOR and LOCATE to move the ball and effect a hit, it is a relatively simple matter to add boundaries, two or more paddles, sound, etc. (Of course, the same result can be accomplished by player/missile graphics, but that's an advanced technique tackled later in this book.)

In the program below, A and B are the X and Y coordinates of the paddle. X and Y are the X and Y coordinates of the ball. C relates to random changes in the color of the paddle. S relates to the speed with which the ball moves.

Program.

```
1 REM * USING COLOR & LOCATE *
2 REM * MICHAEL A. GREENSPAN *
10 S=51:GRAPHICS 3
20 A=35:B=10:X=0:Y=INT(RND(0)*19)+1:C
   =INT(RND(0)*15)+1
25 REM PLOT THE PADDLE
30 COLOR 1:SETCOLOR 0,C,8:PLOT A,B:PL
   OT A,B+1
35 REM MOVE THE PADDLE UP?
40 IF STICK(0)=14 THEN COLOR 4:PLOT A
   ,B:PLOT A,B+1:B=B-1:IF B<0 THEN B=
   0
50 IF STICK(0)=14 THEN GOTO 30
55 REM MOVE THE PADDLE DOWN?
```

```
60 IF STICK(0)=13 THEN COLOR 4:PLOT A
   ,B:PLOT A,B+1:B=B+1:IF B>19 THEN B
   =19
70 IF STICK(0)=13 THEN GOTO 30
75 REM PLOT THE BALL AND HOLD IT AT T
   HAT LOCATION WHILE THE COMPUTER CO
   UNTS FROM 1 TO S
80 COLOR 2:PLOT X,Y:FOR D=1 TO S:NEXT
    D
85 REM CHECK IF THE BALL HIT THE PADD
   LE
90 LOCATE X+1,Y,Z
95 REM MOVE BALL TO THE RIGHT IF IT H
   AS NOT REACHED THE END OF THE ROW
100 IF Z<>1 THEN IF X<=35 THEN COLOR
    4:PLOT X,Y:X=X+1:GOTO 30
105 REM IT'S A MISS
110 IF Z<>1 THEN IF X>35 THEN MISS=MI
    SS+1:? "HITS-";HIT;"  MISSES-";MI
    SS:COLOR 4:FOR B=0 TO 19:PLOT 35,
    B:PLOT 36,B
120 IF Z<>1 THEN NEXT B:S=S+10:GOTO 2
    0
125 REM IT'S A HIT
130 HIT=HIT+1:? "HITS-";HIT;"  MISSES
    -";MISS:S=S-10:COLOR 4:FOR B=0 TO
     19:PLOT 35,B:PLOT 34,B:NEXT B:GO
    TO 20
```

Chapter 2

Customizing The Graphics Modes

How To Design Custom Graphics Modes

Craig Chamberlain

It is well known that the Atari 400/800 computers have superior graphics. One of the things that makes the Atari graphics superior is the fact that the graphics capabilities are flexible. This versatility is demonstrated by the several unique graphics modes that can be generated by the hardware. The Operating System recognizes 12 of these modes, but there are also five other modes available. The table describes some characteristics of the various graphics modes.

There are two varying factors which distinguish one graphics mode from another. First, the pixel size or *resolution* (number of pixels it takes to fill the screen) can differ. Second, the number of color possibilities per pixel may change. The various modes offer different combinations of these two qualities. Because there are so many modes to choose from, it is easier to find one to suit a particular application, which is one reason why Atari graphics are so versatile.

In BASIC, the GRAPHICS command (or GR. in Atari BASIC) is used to change the screen from one graphics mode to another. A number from zero to 11 must follow the GRAPHICS command. This number corresponds to the 12 graphics modes supported by the Operating System. An overview of the general characteristics for these modes is given here.

Operating System Graphics Modes

0	primary text (default mode)
1, 2	color text
3, 5, 7	three-color bit-mapped graphics, various resolutions
4, 6	one-color bit-mapped graphics, various resolutions
8	high-resolution mode, one color
9, 10, 11	specialty modes (explained in other articles)

For all graphics modes except mode zero, a small, four-line text window is provided at the bottom of the screen. If this text window is not desired, it can be eliminated by adding 16 to the number after the GRAPHICS command. Whereas a GRAPHICS 3 changes the screen to mode three with a text window, GRAPHICS 19 changes the screen to mode three with no text window.

Whenever the screen is changed to a new mode using the GRAPHICS command, the screen is automatically cleared, in case any unwanted data might have been left in the screen memory. To defeat this automatic clearing of the screen, add 32 to the number after the GRAPHICS command. This is of little use, however, to BASIC programs.

Using the GRAPHICS command changes the whole screen to a new mode. But is it possible to mix graphics modes? Of course. The text window at the bottom of a screen is actually mode zero combined with the other mode above it. But then, what says that the text window has to be at the bottom of the screen, or that the text must be shown in mode zero? What if it is necessary to use the other graphics modes not supported by the Operating System and BASIC? Doing all these wonderful things requires a little more technical knowledge of Atari graphics, and it starts with something called the *display list*.

When BASIC is given a GRAPHICS command, the Operating System not only reserves room for display data, but also creates a display list. A display list is a sequence of bytes in memory that, among other things, defines the format of the screen.

We'll talk more about the display list, how to find it and how to change it, but first we must delve just a little deeper into Atari graphics terminology.

When you see a screen of a certain graphics mode, you are actually seeing a screen of several identical *mode lines*. A mode line is equivalent to one row of the screen. It is a horizontal strip or section of the screen and is one pixel high. Therefore, the vertical resolution (how many rows) of a graphics mode tells how many mode lines are needed. Each mode line determines the number of pixels and colors that will span from left to right (how many columns).

For example, a mode zero screen offers resolution of 40 across by 24 up and down. In order to produce a mode zero screen, 24 mode lines of mode zero will be required. Each of those mode lines will consist of 40 characters across.

So, the idea of a full-screen graphics mode does not really apply. Rather, a full screen is a bunch of mode lines stacked vertically to fill up the screen.

A mode line is just as high as a pixel, but the actual height of a mode line can vary. The unit used for measuring the height of a mode line is the *scan line.* Just as a screen consists of mode lines, a mode line consists of a certain number of scan lines. Different mode lines have different numbers of scan lines. The table shows how many scan lines are contained in each mode line.

Why all the fuss about scan lines? Because there is a limit to how many scan lines can be displayed on a screen. As a rule, whenever the Operating System creates a screen of any graphics mode, it always uses just the right number of mode lines so that the scan line total equals 192. One hundred ninety-two is the maximum number of scan lines that the average television set can display without excessive *overscan* (cutoff). A screen can have fewer than 192 scan lines without any problem, but to use many more than 192 is only inviting trouble.

Anyway, remember that different mode lines have different numbers of scan lines, and the desirable total scan line count is 192. These two factors control the vertical resolution in a mode as follows:

Given a graphics mode, take 192 scan lines, divide by the number of scan lines per each mode line, and the result is the proper number of mode lines for that particular graphics mode. And, as demonstrated earlier, one mode line corresponds to one horizontal row on the display, so the number of mode lines is the same as the number of rows, which is called vertical resolution.

Now, how was that again? Here's an example using our familiar friend, graphics mode zero. According to the chart, a mode line in mode zero consists of eight scan lines. One hundred ninety-two scan lines divided by eight scan lines per mode line is 24 mode lines. Indeed, the vertical resolution of mode zero is 24 rows.

This is where the display list comes in. The display list describes how many of which mode lines are used to fill the screen from top to bottom. According to our previous example, a display list for a mode zero screen will have to indicate that 24 mode lines of mode zero are to be used. Actually, a mode zero display list looks like this:

Mode Zero Display List

```
112
112
112
 66
XXX
XXX
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
  2
 65
XXX
XXXX
```

It is immediately noticeable that there are no zeroes in the display list. On the other hand, the number two is certainly used often enough. This brings up an important point. The number found in a display list to indicate a mode line is not the same number used by the Operating System for that mode. The table presented at the end has a column marked IR CODE. The label IR stands for *Instruction Register*. The column shows the hardware equivalent (IR number) for all Operating System

modes, as well as for modes not supported by the Operating System. Mode three uses an IR code of eight. IR code four is a multicolor character text mode not normally available. Mode zero is indicated in a display list by an IR number two, which explains the frequent occurrence of that number in the display list example.

The number two, however, is not the only number in the display list example. Now it is time to fully explain the structure of the display list and reveal what the other numbers mean.

The number 112 is used three times at the beginning of the display list. Together, these three numbers tell the video hardware to display 24 empty scan lines at the top of the screen, before the place where the picture starts. These are not mode lines, and do not count as part of the 192 scan lines. Instead, they are called "blank lines," and they create a border at the top of the screen in the background color, just before the 192 scan lines of display. This convention is used by the Atari to reduce overscan problems.

An entry in the display list can show from one to eight blank lines. The number to be used in the display list is derived using the following process:

To show N blank lines, the display list number is (N-1)*16. To show 8 blank lines, 8-1 = 7 and 7*16 = 112, so every use of the number 112 in the display list causes the hardware to show eight scan lines in the background color. Three uses of 112 gives a total of 24 blank lines.

0	1 blank line
16	2 blank lines
32	3 blank lines
48	4 blank lines
64	5 blank lines
80	6 blank lines
96	7 blank lines
112	8 blank lines

The next number in the display list looks like a 66, but it is not a 66. It is a 64 + 2. The 2 indicates that a mode zero mode line should follow the blank lines. The 64 is a "load memory scan counter" (LMS) command, and means that the next two bytes form an address which points to where the display memory (screen data) starts. Because the display data is always put at the top of memory, the two numbers after the 66 will vary on

different computers, according to the amount of RAM installed in each computer.

Since the address of the display memory is broken down into two bytes, a little bit of math will be needed to reconstruct the address. The two bytes are in low-byte, high-byte format. To compute the address, take the high-byte (the second of the two numbers), multiply it by 256, then add it to the low-byte. The result is the address of the first byte of display memory. If a 10 were POKEd at this location on a mode zero screen, the upper leftmost character on the screen would be changed to an asterisk. Adding 20 to the address and doing another POKE will cause an asterisk to appear in the middle of the top row of the screen.

The important point to remember is that by adding 64 to a normal mode line number, in this case a 2, the graphics hardware will not only process the mode line, but perform a LMS command as well. The two bytes immediately following the mode line with the 64 added will form an address that tells the hardware where the following display data resides in memory. The LMS operation actually happens before the mode line starts.

The LMS command is normally used at the beginning of a screen, on the first mode line, but it can be done on any mode line, or on several mode lines, for special applications. Display lists created by the Operating System always have only one LMS command, on the first mode line, except for modes eight through eleven, which for technical reasons require another LMS command in the middle of the display list.

The next numbers in our example display list are a bunch of 2's. There are 23 of them, to be exact. These are the remaining 23 mode lines of mode zero. Remember that the first one was the mode line with the LMS command.

Following the mode lines are a number 65 and a final two bytes. The 65 is another special number which technically means "perform a display list jump and wait for vertical sync." For our purposes, the 65 simply means "this is the end of the display list; go back to the beginning of the display list when the television scanning beam is ready to start drawing another frame." The two bytes after the 65 are in low-byte, high-byte format and represent an address. This address points to the top of the display list. Now would be a good time to tell where the display list is placed. Whenever the Operating System is requested to

create a screen of a certain graphics mode, it always puts the display list just before the display data. So, just as the display memory address varies according to the amount of memory in the computer and the graphics mode, so will the display list address vary.

That concludes the explanation for a normal mode zero display list. It should now be obvious that mixing modes on one screen is just as easy as changing the mode lines in the display list. But first, we need to know how to determine exactly where the display list resides in memory. We know that the address of the beginning of the display list is given at the end of the display list, after the 65, but that won't do us any good if we don't know where the display list is located in the first place.

Fortunately, there is a way to find the address of the beginning of the display list. The same address given in the two bytes after the 65 is also stored in memory locations 560 and 561.

SDLSTL $0230 560 shadow display list address low-byte
SDLSTH $0231 561 shadow display list address high-byte

The address is broken down into two bytes and must be reconstructed using the same procedure shown earlier. In BASIC, the standard method is to use the variable DL for the display list address:

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

After issuing a GRAPHICS 0 command and assigning DL, a PEEK(DL) should return a 112, as will PEEK(DL+1) and PEEK(DL+2). But PEEK(DL+3) will return a 66.

To change mode lines in the display list, POKE statements must be used. For example, a POKE DL+20,4 will put a multicolor text mode line in the middle of the mode zero screen. Try typing on that row and see what happens.

Next, type some characters below the multicolor text mode line, do a POKE DL+20,7, and watch carefully. A mode two line will now be in the middle of the screen, but there will be side effects as well. Two problems will be evident: the bottom of the screen is now a little lower than before, and text below the mode line is not properly aligned.

As for the first problem, a quick glance at the chart will reveal that we replaced a mode zero line of eight scan lines

with a mode two line of 16 scan lines. The display now has more than 192 scan lines, hence the bottom of the screen appears lower.

This problem can be fixed, somewhat. It is necessary to delete the eight extra scan lines, which can be easily done by getting rid of the last mode line. Eliminate the last mode line by executing these instructions:

```
POKE DL+28,65
POKE DL+29,PEEK(560)
POKE DL+30,PEEK(561)
```

All we did was place the "end of display list" command a little earlier in the display list, which effectively cuts off any display below that point. There is a new problem, however, because now there are only 23 rows, but the Operating System still "thinks" there are 24. Hmmm. We traded one problem for another. Let's change the subject and explore the problem of the incorrectly aligned text.

The text below the mode two line has been displaced by 20 characters. The explanation for this is really quite simple. We replaced a mode zero line that needed 40 bytes of data with a mode line that, according to the chart, requires only 20 bytes. There are now an extra 20 bytes on the screen, but the Operating System again is not aware that a change has been made. The text could be realigned with a POKE DL+21,7 but then there would be 40 extra bytes, or essentially, enough for another row.

All of these problems are conflicts with the Operating System. The Operating System establishes the display list but does not monitor it. Changes to the display list only affect the hardware and screen display. Such problems are not always easy to deal with, so they are discussed in separate articles. The key things to remember are:

1. The display should not exceed 192 scan lines.

2. When creating a custom display list, the number of mode lines is limited by the number of mode lines normally allotted for the current mode. (You can mix only up to 24 mode lines on a mode zero screen.)

3. Care must be taken that the mode lines do not require a larger total amount of memory than was designated for the current mode. (Changing several 20-character mode lines to 40-character mode lines would be one way to cause this problem.)

We have now covered the main points of what a display list is, why it is needed, what purpose it serves, how to find it in memory, how to change it, and what problems can be expected as a result of these changes. The display list also controls horizontal and vertical fine scrolling, and a special interrupt, but these are more advanced topics.

To further demonstrate how to modify a display list, three BASIC programs have been provided.

Program 1 prints a display list of mode three with a text window, then changes the bottom text lines to mode one. The display list in this case consists of the 24 blank lines (112,112,112), the LMS command on the first mode three mode line (64+8=72), the address of the display memory, more mode three mode lines (8,8,8,...), another LMS command on the first line of the mode zero text window (64+2=66), the address of the text window memory, the remaining three mode lines of mode zero (2,2,2), and the return (65) followed by the address of the beginning of the display list.

Program 2 creates a mode three screen with a text window, but then moves the text window to the top of the screen. Brief screen flitter is normal.

Program 3 displays from three to 24 blank lines at the top of the screen, then mixes 14 different modes on the screen. Try moving the cursor around and typing in the different modes.

Table. The Graphics Modes.

OS	IR	C	SL	V	H	B
0	2	1	8	24	40	40
–	3	1	10	–	40	40
–	4	4	8	24	40	40
–	5	4	16	12	40	40
1	6	4	8	24	20	20
2	7	4	16	12	20	20
3	8	3	8	24	40	10
4	9	1	4	48	80	10
5	10	3	4	48	80	20
6	11	1	2	96	160	20
–	12	1	1	192	160	20
7	13	3	2	96	160	40
–	14	3	1	192	160	40
8	15	1	1	192	320	40

OS OS MODE
IR IR CODE
C COLORS (PLAYFIELDS)
SL SCAN LINES
V ROWS (MODE LINES)
H COLUMNS
B BYTES

Program 1.

```
100 GRAPHICS 3:REM 3 COLORS, 40X24
110 COLOR 1:REM GOLDEN ORANGE
120 PLOT 0,0
130 DRAWTO 19,19
140 COLOR 2:REM LIGHT GREEN
150 DRAWTO 38,0
160 DL=PEEK(560)+256*PEEK(561)
170 REM DL IS ADDRESS OF DISPLAY LIST
180 FOR K=0 TO 33
190 PRINT "PEEK(DL+";
200 PRINT K;
210 PRINT ")=";
220 PRINT PEEK(DL+K)
230 FOR J=1 TO 333:REM DELAY LOOP
240 NEXT J
250 NEXT K
260 PRINT CHR$(125);:REM CLEAR TEXT
270 PRINT "NOW WATCH THE MODE"
280 PRINT "LINES GET CHANGED"
290 PRINT "AT THE BOTTOM"
300 PRINT "OF THE SCREEN";
310 FOR K=1 TO 999
320 NEXT K
330 POKE DL+25,64+6:REM CHANGE LMS
340 FOR K=1 TO 333
350 NEXT K
360 POKE DL+28,6
370 FOR K=1 TO 333
380 NEXT K
390 POKE DL+29,6
400 FOR K=1 TO 333
410 NEXT K
420 POKE DL+30,6
430 FOR K=1 TO 333
440 NEXT K
450 PRINT
460 PRINT
470 FOR K=1 TO 333
480 NEXT K
490 PRINT CHR$(125);
500 END
```

Program 2.

```
100 GRAPHICS 3
110 DL=PEEK(560)+256*PEEK(561)
120 LMSLO=PEEK(DL+4)
130 LMSHI=PEEK(DL+5)
140 TLO=PEEK(DL+26)
150 THI=PEEK(DL+27)
160 POKE DL+3,64+2
170 POKE DL+4,TLO
180 POKE DL+5,THI
190 FOR K=DL+6 TO DL+6
200 POKE K,2
210 NEXT K
220 POKE DL+9,64+8
230 POKE DL+10,LMSLO
240 POKE DL+11,LMSHI
250 FOR K=DL+12 TO DL+30
260 POKE K,8
270 NEXT K
280 COLOR 1
290 PLOT 0,0
300 DRAWTO 19,19
310 COLOR 2
320 DRAWTO 38,0
330 END
```

Program 3.

```
100 GRAPHICS 0
110 FOR K=1 TO 23
120 PRINT "ATARI ATARI ATARI ATARI"
130 NEXT K
140 DL=PEEK(560)+256*PEEK(561)
150 POKE DL,0
160 POKE DL+1,0
170 POKE DL+2,0
180 FOR K=16 TO 112 STEP 16
190 POKE DL,K
200 FOR J=1 TO 100
210 NEXT J
220 NEXT K
```

```
230 FOR K=16 TO 112 STEP 16
240 POKE DL+1,K
250 FOR J=1 TO 100
260 NEXT J
270 NEXT K
280 FOR K=16 TO 112 STEP 16
290 POKE DL+2,K
300 FOR J=1 TO 100
310 NEXT J
320 NEXT K
330 FOR K=1 TO 23
340 READ P
350 POKE DL+5+K,P
360 FOR J=1 TO 100
370 NEXT J
380 NEXT K
390 DATA 3,4,5,6,7,7
400 DATA 8,8,8,8,9,9,9,9
410 DATA 10,10,11,11
420 DATA 12,12,13,14,15
430 END
```

Put Graphics Modes 1 And 2 At The Bottom Of Your Screen

R. Alan Belke

Most of you who are regular readers of **COMPUTE!** are familiar with the mixing of the graphics modes. The only problem is that you can't use a mode past its regular range. That is, if you wanted to use mode 1 past line 20 or mode 2 past line 10, you couldn't. So you were stuck putting text you wanted at the top of the screen or in the text window. Until now, that is!

What's The Display List?

First we'll look at the "display list" to see what it is and what it does. Figure 1 shows the display list for mode 3. You can verify this by running Program 1. Locations 560, 561 contain the starting address of the list.

Figure 1.

```
112,112,112,72,112,158,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,66,96,159,2,2,2,
  65,78,158
```

The purpose of the list is to tell the computer how to display the information stored in the screen and/or text memories. Let's see how it does this. The first three bytes (112) set up the margin at the top of the screen. Next comes what I call an address byte (72), in this case, a mode 3 address byte. (Figure 2 shows what the address bytes are for each of the modes.) This byte pulls double duty. First, it sets the first line to mode 3. Then it tells the computer that the next two bytes contain the address of the screen memory.

Figure 2.

MODE	0	1	2	3	4	5	6	7	8
ADDRESS BYTE	66	70	71	72	73	74	75	77	79

The next 19 bytes (8) set one line each to mode 3. I call these

mode 3 bytes. You get the value for these bytes by subtracting 64 from the address byte (72-64=8). From this, we can deduce that any byte with bit 6 on is an address byte. Also, notice that 19 mode 3 bytes with the mode 3 address byte give you 20 rows of mode 3, which fills the screen up to the text window.

For whatever mode you are in, you will have one address byte and the number of rows, minus one, regular bytes. For example, mode 7 will have a mode 7 address byte (77) and 79 regular mode 7 bytes, giving you 80 rows. To find out how many rows each mode has, check the "Table of Modes and Screen Formats." It's on the inside back cover of your *Atari BASIC Reference Manual.*

The Last Three Rows Of The Text Window

Now here's the important part. The next byte (66) is a mode 0 address byte. But, instead of the next two lines containing the address of the screen memory, they contain the address of the text editor memory. This is the start of the text window. Modes 1 through 8 use the screen memory. Mode 0 uses the text editor memory. As you may have already guessed, the next three bytes (2) are mode 0 bytes, giving us the last three rows of the text window. If we were in a full screen format, these last six bytes would not be here.

Now we are to the end of the list. This next byte (65) is also an address byte. But it has a special purpose. It tells the computer that it has reached the end of the list and that the next two bytes contain the starting address of the list. (The same as locations 560, 561.)

Before we go on, let me say that the bytes that contain the addresses may vary, depending on the mode you're in and on the amount of memory you have. All the other bytes will be the same.

So how do we get modes 1 and 2 on the bottom of the screen? It's simple! Basically, all we do is change the mode 0 bytes to mode 1 or 2 bytes. Presto! The computer now displays the text editor memory in modes 1 or 2.

Let's look at Program 2 to see how this is done:

Line 10: sets the margins to 40 characters per line and selects mode 3 with text window. Then it finds the address of the display list.

Line 20: searches the list for the start of the text window.

Line 30: changes the mode 0 bytes to mode 1 bytes.

There are a few things to be aware of. Even though you are using modes 1 and 2, you're using the text editor memory; so the computer thinks in 40-column, not 20-column, lines, which means two lines now equal one old line. Here is an example. Suppose we use an empty PRINT statement, planning to leave a blank line. Sorry, it won't work. We would have two blank lines. What we do is put 20 spaces in front of what we want printed on the second line. Also remember that we are using the text editor, so PRINT #6 will not work. Try some different things yourself.

What About Mode Two?

Well, that's almost as simple. Mode 2 lines are twice as wide as modes 1 and 0; so there are only two combinations using mode 2 possible: two rows of mode 2 or one row of mode 2 with two rows of mode 1. We can use only the amount of room that was originally there. Program 3 uses the latter option from above:

Lines 10-20: same as Program 2.

Line 30: basically the same as in Program 2; only this time we make the second line mode 2. And, since we use one less byte, we have to move the end of the list one location forward.

By now you should be able to change the text window into any combination of modes 1 and 2 you want. If you have a program that would work better with the text at the bottom of the screen or the text window as modes 1 or 2, get to work, experiment! Remember, you're the boss.

Program 1.

```
10 GRAPHICS 3:A=PEEK(560)+PEEK(561)*2
   56
20 D=PEEK(A):? D;",";:IF D<>65 THEN A
   =A+1:GOTO 20
30 ? PEEK(A+1);",";PEEK(A+2)
40 GOTO 40
```

Program 2.

```
10 POKE 82,0:GRAPHICS 3:A=PEEK(560)+P
   EEK(561)*256
20 IF PEEK(A)<>66 THEN A=A+1:GOTO 20
30 POKE A,70:POKE A+3,6:POKE A+4,6:PO
   KE A+5,6
40 ? " ATARI AND COMPUTE!      AN UNBE
   ATABLE  "
50 ? "          TEAM          FOUR LINES
    MODE  1"
60 COLOR 2:SETCOLOR 1,10,6:PLOT 17,1:
   DRAWTO 17,10:DRAWTO 9,18
70 PLOT 19,1:DRAWTO 19,18:PLOT 20,1:D
   RAWTO 20,18
80 PLOT 22,1:DRAWTO 22,10:DRAWTO 30,1
   8
90 GOTO 90
```

Program 3.

```
10 POKE 82,0:GRAPHICS 3:A=PEEK(560)+P
   EEK(561)*256
20 IF PEEK(A)<>66 THEN A=A+1:GOTO 20
30 POKE A,70:POKE A+3,7:POKE A+4,6:PO
   KE A+5,65:POKE A+6,PEEK(A+7):POKE
   A+7,PEEK(A+8)
40 ? " ATARI AND COMPUTE!    1 LINE OF
    MODE 2 "
50 ? "  2 LINES OF MODE 1"
60 COLOR 2:SETCOLOR 1,10,6:PLOT 17,1:
   DRAWTO 17,10:DRAWTO 9,18
70 PLOT 19,1:DRAWTO 19,18:PLOT 20,1:D
   RAWTO 20,18
80 PLOT 22,1:DRAWTO 22,10:DRAWTO 30,1
   8
90 GOTO 90
```

Printing Characters In Mixed Graphics Modes

Craig Patchett

One of the problems of custom graphics modes is how to print characters on mode lines that are out of the usual range of that mode. For example, if we design a graphics mode such that the 30th line is mode two, we would get an error message if we attempted to print on that line. This is because the Atari thinks it is in the regular mode two, which allows only twelve lines of characters. We must therefore find another way to put the characters on the screen.

As you may already realize, the screen is just a type of window looking into a part of memory. If you change that memory, what you see on the screen also changes. The solution, therefore, is just to POKE the characters into the memory locations that correspond to the positions on the screen where we want them to appear.

Where Is The Screen In Memory?

Here is how to find the display list in memory:

BEGIN = PEEK(560) + PEEK(561)*256 + 4

But, you may well ask, what does this have to do with the screen memory, or display memory, as we will call it here? It just so happens that the first two memory locations in the display list point to the beginning of display memory in the following fashion:

DISMEM = PEEK(BEGIN) + PEEK(BEGIN + 1)*256

How Do We Calculate The Exact Memory Locations To POKE Into?

Each mode line uses up a certain amount of memory. As you might guess, different modes use different amounts of memory

per line. To be more exact:

MODE	0	1	2	3	4	5	6	7	8
MEM/LINE	40	20	20	10	10	20	20	40	40

So all we have to do is figure out how much memory is used before the mode line that we want to print on, and add that to DISMEM to determine where we want to start POKEing. As an example of how to do this, let's suppose we have a graphics mode with four lines of mode 1, 50 lines of mode seven, three lines of mode four, and three lines of mode two (4*8 + 50*2 + 3*4 + 3*16 = 32 + 100 + 12 + 48 = 192); and we want to print on the second line of mode two. Checking the table above, we go:

4 lines of mode 1 = 4*20 =	80
50 lines of mode 7 = 50*40 =	2000
3 lines of mode 4 = 3*10 =	30
1 line of mode 2 = 1*20 =	20

(remember, we count only the lines *above* the one we want to print on)

For a grand total of: 2130

Therefore, memory location DISMEM + 2130 represents the first character in the second line of mode 2 *for this particular mode*. Memory location DISMEM + 2131 represents the second character, and so on up to DISMEM + 2149 for the 20th character.

We know that POKEing the appropriate value into the appropriate location will cause the desired character to appear at the desired screen location. Since we already know how to determine the appropriate memory location, we now ask:

How Do I Calculate The Appropriate Value For A Character?

It turns out that the value to POKE for a given character corresponds to the order in which the character descriptions are stored in ROM (see "Designing Your Own Character Sets" in Chapter 3). As a quick memory refresher:

ATASCII VALUE	VALUE TO POKE
0-31	64-95
32-95	0-63
96-127	96-127

For reverse characters, just add 128 to the value of the normal character.

My Brain Is In Hibernation; How Do I Convert A Character String To Its Appropriate Values?

I'll leave you with the following self-explanatory subroutine that will take the (predefined) character string PRINTME$ and the starting memory location STARTHERE (also predefined and equal to DISMEM + offset) and POKE PRINTME$ into the appropriate memory locations. Enjoy!

Program.

```
30000 REM This loop will act on each
      character in PRINTME$
30010 FOR ME=1 TO LEN(PRINTME$)
30020 REM Find ATASCII value of chara
      cter
30030 VALUE=ASC(PRINTME$(ME,ME))
30040 REM Subtract 128 temporarily if
       it's a reverse character
30050 VALUE=VALUE-128*(VALUE>127):REM
       See note below
30060 REM Make the appropriate value
      adjustments
30070 VALUE=VALUE+64*(VALUE<32)-32*(V
      ALUE>31 AND VALUE<96)
30080 REM Convert back to reverse if
      necessary
30090 VALUE=VALUE+128*(ASC(PRINTME$(M
      E,ME))>127)
30100 POKE STARTHERE+ME-1,VALUE:REM R
      emember, ME starts at 0, not 1
30110 ? VALUE
30120 REM Go to next character
30130 NEXT ME
30140 REM All done, say goodby
30150 RETURN
```

Note that (condition) equals 1 if the condition is true, 0 if it's not. Thus, X=126:PRINT (X=126) :PRINT(X=127) will print a 1 followed by a 0.

Add A Text Window To GRAPHICS 0

Charles Brannon

The text window can be a useful feature in the graphics modes, enabling a simultaneous text and graphics display. The text window is very similar to a miniature GRAPHICS 0 text screen: all the editor functions are supported, and scrolling and screen clearing are confined to the small four-line window.

This same capability would be useful for a GRAPHICS 0 display. For example, a menu (a list of choices) could be presented in the top 20 or so lines of the screen, and the user's input taken in the lower four lines of the text window. Any errors, such as the user typing editor keys in an INPUT statement, would not interfere with the rest of the screen. Conveniently, any scrolling when caused by a line like this one:

```
150 PRINT "NAME";:INPUT N$:IF LEN(N$)
    >8 THEN PRINT "*TOO LONG*":GOTO 1
    50
```

would not cause the menu above it to scroll as well.

How is all this done? With a single POKE statement. Location 703 normally contains the number 24. If you POKE a four in its place, the cursor is zapped to the bottom of the screen and the text window is in place.

Note that you can't print to the upper part of the screen with PRINT statements; you have to use PRINT#6 as you do with graphics modes 1 and 2. Also, the POSITION statements affect only the upper part of the display; you must use POKEs to position text window output.

Here is an example program to demonstrate the use of the window. It is a simple disk menu program. Notice that you don't need to use PRINT#6 to print to the upper part of the screen until after the POKE 703,4 takes place.

Program.

```
100 REM DEMONSTRATES TEXT WINDOW
110 REM SIMPLE MENU PROGRAM FOR DISK
130 TRAP 150
140 OPEN #1,6,0,"D:*.*":GOTO 160
150 ? "Can't read directory":END
160 GRAPHICS 0:COL=0:POKE 752,1:REM D
    ISABLES CURSOR
170 DIM A$(20),F$(14):TRAP 230
180 INPUT #1;A$
190 POSITION COL,LINE:? A$(1,14)
200 LINE=LINE+1
210 IF LINE>20 THEN COL=COL+13:LINE=0
220 GOTO 180
230 POKE 703,4:REM CREATES TEXT WINDO
    W
240 FOR I=1 TO 100:? I,:NEXT I:REM ON
    LY FOR DEMONSTRATION
250 ? CHR$(125):? "Run which program"
    ;:INPUT A$:REM CHR$(125) ONLY CLE
    ARS WINDOW
260 TRAP 290
270 F$="D:":F$(3)=A$
280 RUN F$
290 ? "Can't RUN ";F$;"."
300 END
```

Mixing Graphics Modes 0 And 8

Douglas Crockford

Normally, GRAPHICS 8 does not allow text to be displayed alongside the graphics. This machine language routine, when added to a BASIC program, provides this useful feature.

Graphics mode 0 is the Atari text mode. It supports uppercase, lowercase, inverse video, and has a position function for placing text anywhere within a 40 by 24 display field. Graphics mode 8 is the Atari high resolution plot mode. It supports the plotting of points and lines in a 320 by 160 (or 192) display field. It would be very nice to use both modes at the same time. The text window is some help, but it confines the plot to the top and the text to the bottom. Modifying the display list provides a partial solution, but it is awkward and doesn't permit the mixing of text and plot on the same line.

A better solution is to use graphics mode 8 and plot the dots that make up the text characters. This can be done very quickly by a 6502 machine language subroutine, which does things in software which are very similar to what the display hardware does 60 times a second.

The subroutine is called with the USR function. It has four arguments:

the horizontal cursor position
the vertical cursor position
the address of the string to be displayed
the length of the string to be displayed.

So, the code

```
GRAPHICS 0
POSITION X,Y
PRINT STRING$;
```

will produce similar results to

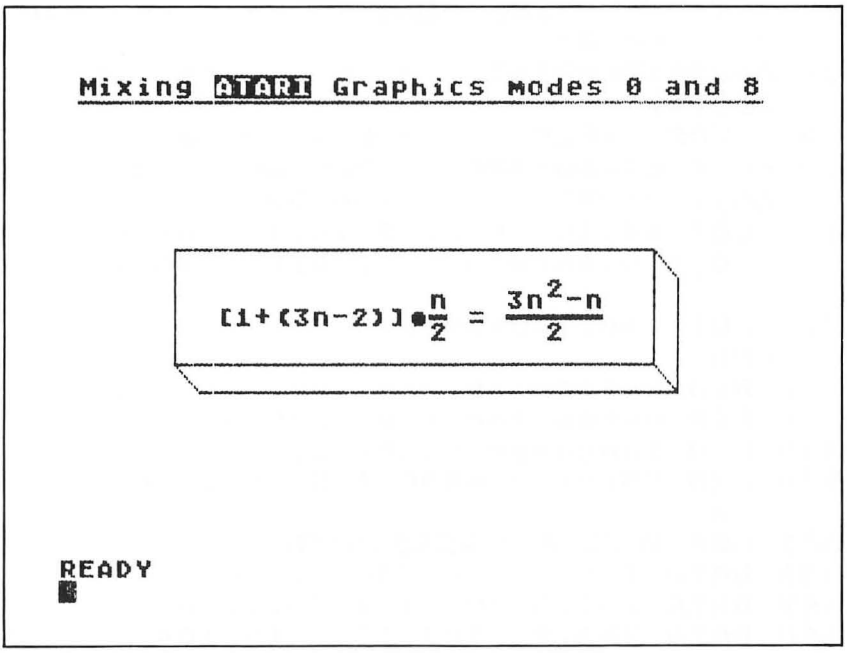
```
GRAPHICS 8
A=USR(ADR(PRINT$),X,Y,ADR(STRING$),LE
N(STRING$))
```

PRINT$ is a string containing the subroutine. The STRING$ should not extend past the last column in a row. Any embedded function codes (cursor movement, insert, etc.) will be displayed literally. The position of the PLOT/DRAWTO pointer is not changed, nor is the current COLOR.

An interesting bonus is that adding 40*R to the horizontal argument causes the text to be displayed R plot rows lower than usual. This permits the display of subscripts, mathematical expressions, 1½ line spacing, underlining, and so on.

The subroutine is relocatable because it contains no JPs, JSRs, or data reference to itself. It can run anywhere in memory. It is also under 256 bytes, so it can also run in page six. The program shows the subroutine being loaded into a string called PRINT$, and shows a few of the things it can do.

Figure. Screen Dump of the Demo Program.



Program.

```
1   REM DEMO
2   REM THIS PROGRAM IS A DEMONSTRATION
3   REM OF MIXING MODES 0 AND 8
10  DIM PRINT$(167)
20  FOR I=1 TO 167
30  READ A:LET PRINT$(I)=CHR$(A)
40  NEXT I
50  GRAPHICS 8
60  MLPRINT=ADR(PRINT$):REM ADDRESS OF
     STRING IS STARTING LOCATION OF MA
    CHINE LANGUAGE ROUTINE
100 A=USR(MLPRINT,10,10,ADR("[1+(3n-2
    )]{T}{R} = {5 R}"),20)
110 A=USR(MLPRINT,101,9,ADR("n"),1)
120 A=USR(MLPRINT,105,9,ADR("3n -n"),
    5)
130 A=USR(MLPRINT,267,8,ADR("2"),1)
140 A=USR(MLPRINT,221,10,ADR("2"),1)
150 A=USR(MLPRINT,227,10,ADR("2"),1)
160 DIM TEXT$(40)
170 TEXT$="Mixing [illegible] Graphics mode
    s 0 and 8"
180 A=USR(MLPRINT,3,0,ADR(TEXT$),LEN(
    TEXT$))
190 COLOR 1:PLOT 24,9:DRAWTO 304,9
200 PLOT 64,60:DRAWTO 260,60:DRAWTO 2
    60,100:DRAWTO 64,100:DRAWTO 64,60
210 PLOT 64,100:DRAWTO 74,110:DRAWTO
    270,110:DRAWTO 270,70:DRAWTO 260,
    60
220 PLOT 260,100:DRAWTO 270,110
230 END
1500 REM Following are the decimal
1510 REM bytes for the machine
1520 REM language routine,
1530 REM "Mixing GRAPHICS Modes 0 AND
      8
1531 REM [illegible]
1536 DATA 104,201,4,240,9,170
1542 DATA 240,5,104,104,202,208
1548 DATA 251,96,104,133,215,104
```

```
1554 DATA 133,214,104,104,168,104
1560 DATA 133,217,104,133,216,104
1566 DATA 104,240,236,133,212,24
1572 DATA 165,214,101,88,133,214
1578 DATA 165,89,101,215,133,215
1584 DATA 152,240,15,165,214,105
1590 DATA 64,133,214,165,215,105
1596 DATA 1,133,215,136,208,241
1602 DATA 132,221,160,0,132,220
1608 DATA 177,216,160,0,170,16
1614 DATA 1,136,132,213,138,41
1620 DATA 96,208,4,169,64,16
1626 DATA 14,201,32,208,4,169
1632 DATA 0,16,6,201,64,208
1638 DATA 2,169,32,133,218,138
1644 DATA 41,31,5,218,133,218
1650 DATA 169,0,162,3,6,218
1656 DATA 42,202,208,250,109,244
1662 DATA 2,133,219,164,221,177
1668 DATA 218,69,213,164,220,145
1674 DATA 214,200,132,220,196,212
1680 DATA 208,182,24,165,214,105
1686 DATA 40,133,214,144,2,230
1692 DATA 215,230,221,169,8,197
1698 DATA 221,208,159,96,207,96
```

Chapter 3

Redefining Character Sets

Designing Your Own Character Sets

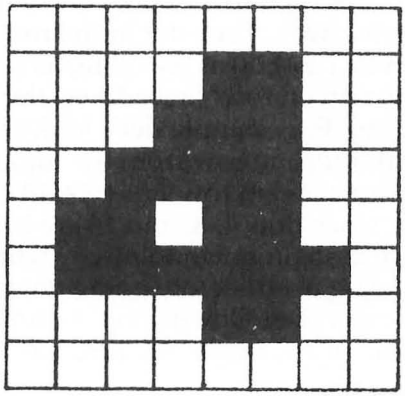
Craig Patchett

You can change the shapes of your characters with this technique.

If you want to draw boxes, or design a card game, then Atari's graphics characters are terrific. But what if you're writing an outer space game or a music program? Wouldn't you prefer a rocket ship or a musical note to a vertical line? This article will explain not only how to change Atari's graphics characters to whatever you desire, but also how to change any Atari character at all, from letters to numbers to punctuation.

How Characters Are Made

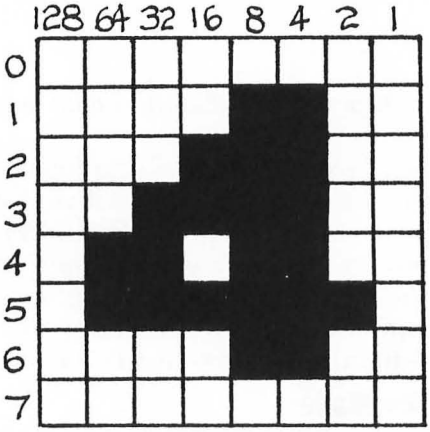
An Atari character, as you may already know, is made up of a bunch of small dots grouped close together. A total of 64 dots, arranged in an eight-by-eight square, can be used to make one character. An Atari "4", for example, really looks like this:



Here, the squares colored in represent the dots that are used. Notice that the outside squares are not used. If they were, then the characters would touch each other when printed side

by side and would be difficult to read. Graphics characters can be made to touch, however, since side by side they could be made to look like one large, continuous character.

Somewhere in memory the Atari has a list of which dots are used for each character. Before we find out where this list is, let's see how the Atari represents each character in the list.



The Atari remembers each character as eight numbers, each representing a row of eight dots. These rows I have numbered above from 0 to 7. Row 0 is always the first number, row 7 the last. The Atari changes each row of dots into a number from 0 to 255 in the following way. Each dot in the row is assigned a multiple of two (from 1 to 128) as its value, as shown above. To get the number for a given row, just add up the values of the dots used in that row. For example, let's look at the "4". The number in row 1 (the second row from the top) will be 12, since dots 4 and 8 are being used in row 1 (4+8=12). The third number will be 28, since dots 4, 8, and 16 are being used in row 2 (4+8+16=28), and so on down to row 7, which will be 0, since no dots are being used. Before going on, make sure you understand how to get the following eight numbers as representing the number "4": 0, 12, 28, 60, 108, 126, 12, 0.

How Characters Are Stored

Since there are a total of 128 Atari characters, not counting reverse characters (see Appendix C: ATASCII Character Set, in the *Atari BASIC Reference Manual*), the list will contain 1024 numbers (8 numbers per character times 128 characters =1024

numbers). Look at Appendix D: Atari 400/800 Memory Map in the *BASIC Reference Manual*. This simply describes what some of the different memory locations are used for. We're interested in the first locations, containing the "Operating System ROM." The Operating System is just a built-in program that tells the Atari how to do everything it can do in the "Memo Pad" mode, simple things such as putting a character on the screen when a key is pressed, etc. ROM (Read Only Memory) means that the program will always be in the computer's memory, even when the computer is turned off, and can never be changed by the programmer (that's you). Unfortunately, the first 1024 locations in the Operating System ROM (locations 57344 to 58367) contain the list of numbers we are interested in. In order to change the characters we are going to have to change the list, which ROM won't let us do. There's an easy way out, however, and that's to move the list to a place where we can change it.

Protecting The Character List

We need a place where the list will be safe from our accidentally changing it, but where we will be able to change it when we want to. Look at Appendix D again. About halfway down the page is a box labeled "RAMTOP." RAMTOP points to the last location in user memory, the memory we have available for our use. What if we were to change RAMTOP so that it pointed 1024 locations before the end of user memory? Then the Atari would think that user memory ended at the new RAMTOP and would not try to put anything in memory after that location. We would still be able to use those locations ourselves, though.

Let's flip over to Appendix I: Memory Locations. If we look up decimal location 106, we see that it contains the value of RAMTOP. So if we change location 106, we can trick the Atari into staying away from our list. Before we do that, however, let me point out that adding one to the value in 106 actually adds 256 to RAMTOP. This is because of something called "paging," which is too complicated to explain here, and not really important for what we're doing anyway. Just be aware that to move RAMTOP back 1024 locations, we need to subtract four (4 x 256 = 1024) from location 106. To give us some extra space in case the Atari accidentally goes a little past RAMTOP, we'll subtract five instead. We do this using POKE and PEEK (finally some programming!):

```
10 POKE 106, PEEK (106)-5:GRAPHICS 0
```

The reason we use a GRAPHICS 0 right after changing RAMTOP is because the Atari normally stores screen data in the locations we'll be using for the list (see Chapter 6). If we don't use a GRAPHICS command to move that list to a new location, the screen will do strange things when we move the character list into place, which we are now ready to do.

Relocating The List

Moving the list is extremely simple; we just use a FOR/NEXT loop and POKE the values from ROM into their new locations. We first need to figure out the value of the location of the first number in the new list:

```
20 STARTLIST=(PEEK(106)+1)*256
```

Remember, we subtracted an extra one from location 106 to be safe, so we have to add it back on to determine the start of the list. Also, don't forget that we have to multiply the value in 106 by 256 because of paging. Now let's move:

```
30 ? "HOLD ON...":FOR MOVEME= 0 TO 1023:POKE
STARTLIST+MOVEME,PEEK(57344+MOVEME): NEXT
MOVEME
```

All that's left now is to tell the Atari where the new list is. We do this by changing the value in location 756, which points to the starting location of the character set to be used (look at Appendix I). If you look at location 756 at this stage (use PRINT PEEK(756)), you'll see that it contains the value 224. Again, because of paging, this really means 224 x 256, or 57344 (surprise!), the starting location of the character set in ROM. So we go:

```
40 POKE 756,STARTLIST/256
```

A few words of warning about location 756. Every time you use the GRAPHICS command, the Atari sets the value in location 756 back to 224. That means that after each GRAPHICS command, you'll have to execute the equivalent of line 40. No big deal, but if you forget....

Redefining Characters

Before we actually make any changes, let's look at the order the characters are stored in the list. For this we'll need Appendix C again (and you thought you'd never use the Appendices!). Unfortunately, Atari chose not to store the characters in memory

exactly in the ATASCII order. Almost but not exactly:

TYPE	ATASCII ORDER	MEMORY ORDER
uppercase, numbers, punctuation	32-95	0-63
graphics characters	0-31	64-95
lowercase, some graphics	96-127	96-127

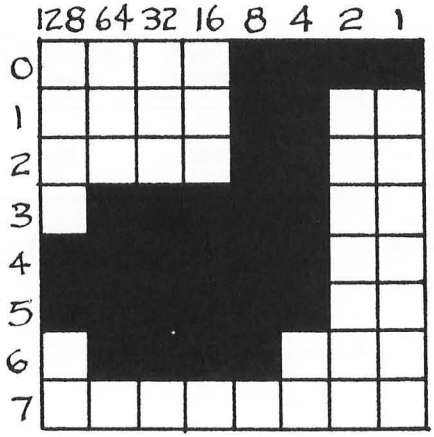
As you can see, all that Atari did was to move the graphics characters between the uppercase and lowercase (they did this in order to be able to choose between uppercase and lowercase/graphics in modes one and two). In the meantime, they made our job harder for us. In order to determine where a character is stored in memory, we have to perform a *little* mathematical wizardry on its ATASCII value. In the following "formulas," keep in mind that each character is represented by eight numbers, which is why we multiply by eight:

ATASCII VALUE (AV)	MEMORY LOCATION (of first number)
32-95	(AV-32)*8 + STARTLIST
0-31	(AV + 64)*8 + STARTLIST
96-127	AV*8 + STARTLIST

Of course, to get the location of the original character (in ROM), we would add 57344 instead of STARTLIST.

With these mathematical manipulations in mind, let's try one of the original examples that I mentioned. We'll change one of the graphics characters, let's say ◀CTRL▶T, to a musical

note. First, let's design our note:



This may not look exactly like a note as is, but because of the size of the dots, it will look fine when printed on the screen, as we shall soon see. I'll leave it up to you to check for yourself that the note translates into the following eight numbers: 15, 12, 12, 124, 252, 252, 120, 0. We now want to replace the eight numbers already in memory for ◀CTRL▶T with these eight. ◀CTRL▶T has an ATASCII value of 20 (see Appendix C), which fits in the 0-31 category in the formula chart above. The first thing to do, therefore, is to add 64 (20 + 64 = 84) and multiply by eight (8x84 = 672) to give us a value of 672. So to change the ◀CTRL▶T character we would have to change the eight numbers in memory beginning with location 672 + STARTLIST. We make this change using a FOR/NEXT loop and DATA statements:

```
50 FOR MOVEME=0 TO 7:READ VALUE:POKE 672+
   STARTLIST+MOVEME, VALUE:NEXT MOVEME
60 DATA 15, 12, 12, 124, 252, 252, 120, 0
```

Now, after this has been RUN, whenever we use a ◀CTRL▶T, we will have a musical note. Try it!

As an informal kind of self-test, make sure you understand the following two lines. Try to work out which character they will change, and what the new character will look like, before you actually RUN them (with the rest of the program, of course):

```
70 FOR MOVEME=0 TO 7:READ VALUE:POKE 776+
   STARTLIST+MOVEME, VALUE:NEXT MOVEME
80 DATA 0, 0, 60, 102, 102, 102, 63, 0
```

As you can see, lines 50 and 70 are very much alike except for the initial value added to STARTLIST. This should light up a sign in your brain saying "SUBROUTINE!" If you have more than one or two characters to be redesigned, you should use a subroutine to save memory.

A Few Details And Programming Hints

- In graphics modes one and two, to use lowercase and graphics characters with your new character set, POKE 756 with STARTLIST/256+2. To go back to uppercase, etc., POKE 756 with STARTLIST/256.
- If you press the RESET button, the Atari will change the value of location 106 and put the display list back in place of your character set. Under such circumstances it is necessary to run the program over again in order to get your character set back.
- If a character is too complicated to put in an eight by eight box, then use more than one box (and therefore more than one character), and combine them in a string. For example, using the Atari's regular graphics characters:

```
DIM BOX$(7):BOX$="(see below)":PRINT BOX$
Type BOX$ as ◀CTRL▶Q, ◀CTRL▶E, ◀ESC▶ ◀CTRL▶
=, ◀ESC▶ ◀CTRL▶+, ◀CTRL▶Z, ◀CTRL▶C.
```

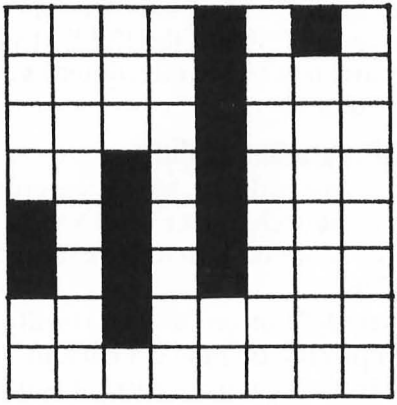
Bonus: Four Colors In Graphics Mode 0!

It is possible to define a character to be one of three different colors (4=3+background). The only drawback is that once you have defined the letter "A" to be orange, for example, all A's will be orange, not just the ones you want.

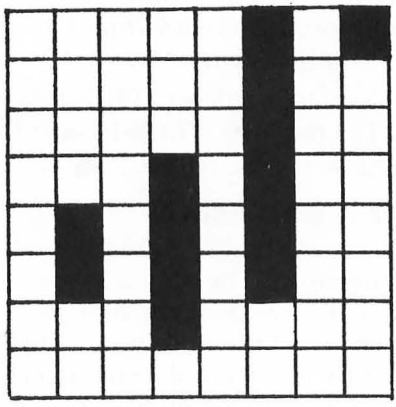
How do we define the color of a character? It's really quite simple. Just as in graphics mode eight, a dot in an even-numbered column will be a different color than a dot in an odd-numbered column. Two dots side by side will produce yet another color. This is why an Atari "4" (and all other Atari characters) and my musical note have vertical lines that are two dots wide, compared to the horizontal lines that are only one dot wide (or thick, if you prefer). If the vertical lines were only one dot wide, they would be a different color than the horizontal ones, unless the horizontal lines alternated one dot on and one dot off. Confused? Don't worry, just substitute the following

variations of the musical note for the data in the sample program and see what they look like:

60 DATA 10,8,8,40,168,168,32,0



80 DATA 5,4,4,20,84,84,16,0



Such characters will, of course, look unusual in graphics modes one and two, just as they look unusual in the above diagrams.

You can't do a lot of experimenting with this "phenomenon" to get such effects as multicolored characters. Changing the background color will change the colors of the columns, and thus the colors of the characters. Finally, if you need only one "A", or whatever, to be a different color, define it as a graphics character.

Bonus: Upper- And Lowercase In Graphics Modes 1 And 2

By now, after running things over in your mind, you might already suspect how to mix upper- and lowercase in modes one and two. If not, it is a painfully simple trick. Since modes one and two allow use of lowercase and graphics characters together, just redefine the graphics characters to be uppercase letters! You can do this by moving the uppercase character descriptions from the ROM list to your own list, like so:

```
35 FOR MOVEME=256 TO 472:POKE STARTLIST+MOVEME
   +256,PEEK(57344+MOVEME):NEXT MOVEME
```

Typing a ◀CTRL▶A will now give us an uppercase "A" and so on. Of course, this is not the best way to do it, since we no longer have any graphics characters. If we know that we will need only certain uppercase letters in our program, then it would be better to move just those letters, one by one, using the tables given earlier in the article. In any case, we are now able to mix almost any combination of characters we wish in graphics modes one and two.

SuperFont

Charles Brannon

The ability to redefine the character set is one of the more useful features of the Atari. Basically, it involves the plotting of a character on an eight by eight matrix and then converting each row into a binary number.

This process, however, is slow and tedious for the programmer. Fortunately, it is an obvious candidate for computerization. The computer could display a grid, let you set and clear points on it, and then do the binary-to-decimal conversion for you. It could also let you save and load completed *fonts* (character sets) from tape or disk.

Although "SuperFont" may lack some of the features of commercial products, it is quite powerful and versatile. SuperFont is written in BASIC, but what makes it special is that it has several machine language subroutines as well. One of these, thanks to a display list interrupt (DLI), enables the redefined character set to be displayed on the screen at the same time as the regular one. This permits you to see the effects of your changes without letting the command menu or prompts turn into starships.

SuperFont uses player/missile graphics for fast updates and a colorful grid. Since the special character window is set off in a different color than the rest of the screen (again via DLI's), you get eight different colors to delight the eye. The human interface is enhanced with the use of a joystick to plot points in the eight by eight grid.

SuperFont has 18 commands:

This menu is displayed on the screen along with a checkerboard plotting grid, the 128 characters of the character set, and the 128 characters of the alternate character set. Some commands require you to select a character. A cursor will be placed on each of the character sets. You can move the cursors around the sets simultaneously. When the cursor is on the desired character, press the fire button to indicate it. An explanation of each command follows:

Edit: The basic editing command. The selected character is copied into the grid, and a flashing cursor is homed into the grid. You move the cursor with the joystick. Pressing fire will set a point (if a point is clear) or reset (clear) a point (if a point is already set). You can draw lines by holding down the button while moving the joystick. Any changes are immediately visible in the character set and in the character displayed in the graphics mode one and two lines at the bottom of the screen. To completely redesign a character, use the Clear command, and then design the character from scratch.

Restore: This command will "fix" a character by copying the original

bit pattern into it. Very useful if you have mangled a character or changed the wrong one.

Copy From: You select a character which is copied into the current character. The grid is updated, and you can further edit the character.

Copy To: The current character is copied to (replaces) the indicated character.

Switch: Exchanges the current character's bit pattern with the selected character.

Overlay: The selected character is overlaid upon the current character. This lets you combine two characters to form a new one.

Clear: Clears out the current character. For creating unique characters.

Invert: Turns the current character "upside down." For example, a redefined M could be inverted and copied to the W.

Save Font: Saves the alternate character set in compact form with a machine language routine. Answer "Filename?" with either C: or D:filespec. If you see an error message, press a key to return to the menu.

Load Font: Retrieves a character set from tape or disk. Answer the "Filename" prompt as you did in Save Font.

Cursor-up or SHIFT-DELETE: Similar to Delete Line in BASIC. The line of dots the cursor rests on is deleted; the following lines are pulled up to fill the gap.

Cursor-down or SHIFT-INSERT: Similar to Insert Line in BASIC. A blank line is inserted at the cursor position. The bottom line is lost.

Scroll Left: The bit pattern of the character is shifted to the left.

Scroll Right: The bit pattern of the character is shifted to the right.

Write Data: The internal code (0-127) of the current character is printed in reverse-field followed by the eight bytes (in decimal) of the character. If you want a printout of the entire character set, use the auxiliary program CHPRINT (Program 3). Pressing any menu selection key will erase the nine bytes.

Print Character: If you want a hardcopy printout of a character, or only want to define a few characters and have a printed record, this command comes in handy. Just have your printer on-line and press P to create a "picture" of the character (printed

with X's). Beside each line of the character is the decimal value of the binary byte for that line. The number printed at the top is the internal character code for the character you redefined.

Graphics: Toggles the TEXT/GRAPHICS option of the graphics mode one and two lines to let you see each half of the character set.

Reverse: Puts the character in reverse field: all dots become blanks, and all blanks become dots. Reverse field versions of the characters are not normally stored in the character set, but you may want this for special graphics, such as reverse-field text in graphics modes one or two.

Quit: Exits program.

The commands offer flexibility in working with character sets, but there may be other functions you want to add. The program is modular in structure; just follow the branching IF statements after line 790 to 1370 and replace the 520 (IF K <> ASC("G") THEN 520) with a link to your additional command(s). You may also want to change the colors. Besides the SETCOLOR statements in line 170, change the zero in line 300 (POKE 1538,0) to COLOR (0-15)*16 + LUMINANCE (0-14). Similarly, you can play with the player/missile colors in line 360.

It is also possible to use the character set data on tape or disk directly. It is written as a series of 1024 bytes: the bytes of the character set – no more, no less. I have included three extra utility programs which access the character data. Program 2 simply loads the set into memory and changes CHBASE (756) to point to it. Program 3 produces a formatted hex or decimal dump of the character set. Both programs should have the "filespec" changed to the filename of your character set.

You can use Program 4, CHSET DATAMAKER, to create a "module" of lines that lets you add your character set to any program. After saving your character set to tape or disk, just RUN Program 4. It will ask you for the filename of the character set, the starting line number of the "module," and a filename for the module. (Just answer C: to the filename prompt for use with a cassette.)

CHSET DATAMAKER "writes" a subroutine replete with the appropriate PEEKs, loops, and DATA statements, to tape or disk. It optimizes space by writing DATA statements only for those characters you have changed, by comparing your character set to the ROM default character set. After it has

finished, you can merge the lines it produced with any program using the ENTER command. You may need to make some minor adjustments to the code it produces. And in your main program, remember to use a POKE 756,CHSET/256 after every GRAPHICS statement, since a GRAPHICS statement resets the character pointer.

The code of the main program is fairly straightforward. It uses several machine language subroutines: (1) A Display List Interrupt handler to maintain the special character window. (2) Copies the ROM character table into the RAM CHSET table (avoids the 15-second delay in BASIC). (3) A LOGIC subroutine that permits AND, OR, EOR to be used on a binary level (see "Make Your Atari a Bit Wiser," **COMPUTE!**, May 1981, #12, p. 74). (4) Implements a fast machine language memory save, thanks to the Input/Output Control Block (IOCB) PUTREC and GETREC commands.

You can do a lot with this capability: custom fonts (Greek, "computeristic," script), graphics characters (special line drawing characters, spaceships, "invaders," bombs, tanks, planes, ships, even little people). SuperFont makes your task easier, even fun!

Program 1.

```
140 DIM I(7),FN$(14),N$(3)
150 IF PEEK(1536)<>72 THEN GOSUB 1400
160 GRAPHICS 0:POKE 752,1
170 SETCOLOR 2,7,2:SETCOLOR 4,7,2
180 DL=PEEK(560)+256*PEEK(561)+4
190 SD=PEEK(88)+256*PEEK(89)+13*40:AS
    D=SD+5*40
200 A1=1630:FUNC=1631:A2=1632:LOGIC=1
    628
210 RAM=PEEK(106)-8:PMBASE=RAM*256
220 CHRORG=57344
230 POKE 559,46:POKE 54279,RAM
240 POKE 53277,3:POKE 53256,3
250 CHSET=(RAM-8)*256
260 POKE DL+23,6:POKE DL+24,7
270 POKE DL+18,130
280 POKE 512,0:POKE 513,6
290 POKE 54286,192
300 POKE 1549,RAM-8:POKE 1538,0
310 A=USR(1555,CHSET)
320 P0=PMBASE+512+20:P1=PMBASE+640+20
    :P2=PMBASE+768+20:P=PMBASE+896+20
    :T=85
325 FOR I=0 TO 128:POKE P0+I,0:POKE P
    1+I,0:POKE P2+I,0:NEXT I
330 FOR I=0 TO 7:FOR J=0 TO 3:T=255-T
    :POKE P0+I*4+J,0:POKE P1+I*4+J,T:
    T=255-T
340 POKE P2+I*4+J,T:NEXT J:T=255-T:NE
    XT I
350 POKE 53248,64:POKE 53249,64:POKE
    53250,64
360 POKE 704,198:POKE 705,240:POKE 70
    6,68
370 POKE 53256,3:POKE 53257,3:POKE 53
    258,3:POKE 623,1
380 ? " {Q}{8 R}{E}":FOR I=1 TO 8:? "
     |{8 SPACES}|":NEXT I:? " {Z}{8 R}
    {C}"
390 POKE 82,14:POSITION 14,1
400 ? "⊟ Edit{8 SPACES}⊡ Restore"
```

```
410 ? "F Copy From{3 SPACES}X Switch"
420 ? "T Copy To{5 SPACES}C Clear"
430 ? "O Overlay{5 SPACES}I Invert"
440 ? "S Save Font{3 SPACES}L Load Fo
    nt"
450 ? "{ESC}{DEL LINE} Delete
    {6 SPACES}{ESC}{INS LINE} Insert"
460 ? "{ESC}{CLR TAB} Scroll Left
    {ESC}{SET TAB} Scroll{DOWN}
    {6 LEFT}Right"
465 ? "{UP}P Print Char."
470 ? "W Write Data  Q uit"
480 ? "{DOWN}{F}{B}{R} Reverse
    {3 SPACES}G Graphics"
490 FOR I=0 TO 3:FOR J=0 TO 31:Z1=J+I
    *40+4:Z2=I*32+J:POKE SD+Z1,Z2:POK
    E ASD+Z1,Z2:NEXT J:NEXT I
500 POKE 82,2:POSITION 0,0
510 OPEN #2,4,0,"K:"
520 P=PEEK(764):IF P=255 THEN 520
530 IF P=60 THEN 520
540 IF P=39 THEN POKE 764,168
550 GET #2,K
560 IF K<>ASC("E") THEN 790
570 GOSUB 1750
580 FOR I=0 TO 7:A=PEEK(CHSET+C*8+I):
    FOR J=0 TO 3:POKE P0+I*4+J,A:NEXT
     J:NEXT I
590 POKE ASD+169,C:POKE ASD+190,C
600 JX=0:JY=0
610 POSITION JX+4,JY+1
620 ? CHR$(32+128*FF);"{LEFT}";:FF=1-
    FF
630 IF STRIG(0)=0 THEN 750
640 IF PEEK(764)<255 THEN ? " ";:GOTO
     520
650 ST=STICK(0):IF ST=15 THEN 620
660 IF STRIG(0) THEN FOR I=0 TO 100 S
    TEP 20:SOUND 0,100-I,10,8:NEXT I
670 POSITION JX+4,JY+1:? " ";
680 JX=JX+(ST=7)-(ST=11)
690 JY=JY+(ST=13)-(ST=14)
700 IF JX<0 THEN JX=7
```

```
710 IF JX>7 THEN JX=0
720 IF JY<0 THEN JY=7
730 IF JY>7 THEN JY=0
740 GOTO 610
750 POKE A1,PEEK(CHSET+C*8+JY):POKE A
    2,2^(7-JX):POKE FUNC,73:A=USR(LOG
    IC)
760 POKE CHSET+C*8+JY,A:FOR J=0 TO 3:
    POKE P0+JY*4+J,A:NEXT J
770 FOR I=1 TO 10:SOUND 0,I*4,8,8:NEX
    T I:SOUND 0,0,0,0
780 GOTO 650
790 IF K<>ASC("F") THEN 830
800 S=C:GOSUB 1750
810 FOR I=0 TO 7:A=PEEK(CHSET+C*8+I):
    POKE CHSET+S*8+I,A:NEXT I
820 C=S:GOTO 580
830 IF K<>ASC("T") THEN 870
840 S=C:GOSUB 1750
850 FOR I=0 TO 7:A=PEEK(CHSET+S*8+I):
    POKE CHSET+C*8+I,A:NEXT I
860 C=S:GOTO 600
870 IF K<>ASC("O") THEN 920
880 S=C:GOSUB 1750
890 FOR I=0 TO 7:POKE A1,PEEK(CHSET+C
    *8+I):POKE A2,PEEK(CHSET+S*8+I):P
    OKE FUNC,9:A=USR(LOGIC)
900 POKE CHSET+S*8+I,A:NEXT I
910 C=S:GOTO 580
920 IF K<>ASC("R") THEN 940
930 FOR I=0 TO 7:POKE CHSET+C*8+I,PEE
    K(CHRORG+C*8+I):NEXT I:GOTO 580
940 IF K<>ASC("C") THEN 960
950 FOR I=0 TO 7:POKE CHSET+C*8+I,0:N
    EXT I:GOTO 580
960 IF K<>ASC("{R3") THEN 980
970 FOR I=0 TO 7:POKE CHSET+C*8+I,255
    -PEEK(CHSET+C*8+I):NEXT I:GOTO 58
    0
980 IF K<>ASC("X") THEN 1010
990 S=C:GOSUB 1750
1000 FOR I=0 TO 7:A=PEEK(CHSET+S*8+I)
     :POKE CHSET+S*8+I,PEEK(CHSET+C*8
```

```
     +I):POKE CHSET+C*8+I,A:NEXT I:GO
     TO 580
1010 IF K<>ASC("I") THEN 1030
1020 FOR I=0 TO 7:I(I)=PEEK(CHSET+C*8
     +I):NEXT I:FOR I=0 TO 7:POKE CHS
     ET+C*8+I,I(7-I):NEXT I:GOTO 580
1030 IF K<>ASC("{UP}") AND K<>ASC("
     {DEL LINE}") THEN 1050
1040 FOR I=JY TO 6:POKE CHSET+C*8+I,P
     EEK(CHSET+C*8+I+1):NEXT I:POKE C
     HSET+C*8+7,0:GOTO 580
1050 IF K<>ASC("{DOWN}") AND K<>ASC("
     {INS LINE}") THEN 1070
1060 FOR I=7 TO JY STEP -1:POKE CHSET
     +C*8+I,PEEK(CHSET+C*8+I-1):NEXT
     I:POKE CHSET+C*8+JY,0:GOTO 580
1070 IF K<>ASC("{LEFT}") THEN 1100
1080 FOR I=0 TO 7:A=PEEK(CHSET+C*8+I)
     *2:IF A>255 THEN A=A-256
1090 POKE CHSET+C*8+I,A:NEXT I:GOTO 5
     80
1100 IF K<>ASC("{RIGHT}") THEN 1130
1110 FOR I=0 TO 7:A=INT(PEEK(CHSET+C*
     8+I)/2)
1120 POKE CHSET+C*8+I,A:NEXT I:GOTO 5
     80
1130 IF K<>ASC("Q") THEN 1150
1140 POKE 53248,0:POKE 53249,0:POKE 5
     3250,0:POKE 53277,0:GRAPHICS 0:E
     ND
1150 IF K<>ASC("S") THEN 1210
1160 GOSUB 1610:POKE 195,0
1170 TRAP 1190:OPEN #1,8,0,FN$
1180 A=USR(1589,CHSET)
1190 CLOSE #1:TRAP 40000:IF PEEK(195)
      THEN 1260
1200 POKE 54286,192:GOTO 580
1210 IF K<>ASC("L") THEN 1290
1220 GOSUB 1610:POKE 195,0
1230 TRAP 1250:OPEN #1,4,0,FN$
1240 A=USR(1619,CHSET)
1250 CLOSE #1:TRAP 40000:IF PEEK(195)
     =0 THEN 1200
```

```
1260 POSITION 14,0:? "{BELL}* ERROR -
     ";PEEK(195);" *":CLOSE #1
1270 IF PEEK(764)<255 THEN POSITION 1
     4,0:? "{20 SPACES}":GOTO 1200
1280 GOTO 1270
1290 IF K<>ASC("W") THEN 1370
1300 POSITION 2,10:N$="{3 SPACES}":L=
     LEN(STR$(C)):N$(1,L)=STR$(C):L=L
     EN(N$)
1310 FOR I=1 TO L:? CHR$(ASC(N$(I,I))
     +128);:NEXT I:? ">";
1320 FOR I=0 TO 2:FOR J=0 TO 1+(I>0):
     A=PEEK(CHSET+C*8+J+I*3)
1330 SOUND 0,(I*3+J)*10+50,10,8
1340 PRINT A;",";:NEXT J:? "{BACK S}"
     :NEXT I:SOUND 0,0,0,0
1350 IF PEEK(764)=255 THEN 1350
1360 POSITION 2,10:FOR I=1 TO 3:? "
     {12 SPACES}":NEXT I:GOTO 520
1370 IF K<>ASC("G") THEN 1395
1380 CF=1-CF:POKE 1549,RAM-8+2*CF
1390 GOTO 520
1395 IF K<>ASC("P") THEN 520
1397 GOTO 5000
1400 GRAPHICS 2+16:SETCOLOR 4,1,6:POS
     ITION 5,3:? #6;"SUPERFONT"
1410 POSITION 4,5:? #6;"patience{3 N}
     ":POSITION 2,7:? #6;"CHARGES■CRE
     ATED"
1420 FOR I=1536 TO 1639:READ A:POKE I
     ,A:POKE 709,A:SOUND 0,A,10,4:NEX
     T I
1430 SOUND 0,0,0,0:RETURN
1440 DATA 72,169,100,141,10,210
1450 DATA 141,24,208,141,26,208
1460 DATA 169,6,141,9,212,104
1470 DATA 64,104,104,133,204,104
1480 DATA 133,203,169,0,133,205
1490 DATA 169,224,133,206,162,4
1500 DATA 160,0,177,205,145,203
1510 DATA 200,208,249,230,204,230
1520 DATA 206,202,208,240,96,104
1530 DATA 162,16,169,9,157,66
1540 DATA 3,104,157,69,3,104
```

```
1550 DATA 157,68,3,169,0,157
1560 DATA 72,3,169,4,157,73
1570 DATA 3,32,86,228,96,104
1580 DATA 162,16,169,5,76,58
1590 DATA 6,9,104,169,0,9,0,133
1600 DATA 212,169,0,133,213,96
1610 POSITION 14,0:? "Filename?";
1620 FN$="":K=0
1630 POKE 20,0
1640 IF PEEK(764)<255 AND PEEK(764)<>
     39 AND PEEK(764)<>60 THEN 1670
1650 IF PEEK(20)<10 THEN 1640
1660 ? CHR$(21+11*K);"{LEFT}";:K=1-K:
     GOTO 1630
1670 GET #2,A
1680 IF A=155 THEN ? " ";:FOR I=1 TO
     LEN(FN$)+10:? "{BACK S}";:NEXT I
     :RETURN
1690 IF A=126 AND LEN(FN$)>1 THEN FN$
     =FN$(1,LEN(FN$)-1):? " {LEFT}";C
     HR$(A);:GOTO 1630
1695 IF A=126 AND LEN(FN$)=1 THEN ? C
     HR$(A);:GOTO 1620
1700 IF A=58 OR (A>=48 AND A<=57) OR
     (A>=65 AND A<=90) OR A=46 THEN 1
     720
1710 GOTO 1630
1720 IF LEN(FN$)<14 THEN FN$(LEN(FN$)
     +1)=CHR$(A):? CHR$(A);
1730 GOTO 1630
1740 END
1750 REM GET CHOICE OF CHARACTER
1760 CY=INT(MRY/32):CX=MRY-32*CY
1770 C=CX+CY*32
1780 POKE SD+CX+CY*40+4,C+128
1790 POKE ASD+CX+CY*40+4,C+128
1800 IF STRIG(0)=0 OR PEEK(764)<255 T
     HEN MRY=C:GOTO 1900
1810 ST=STICK(0):IF ST=15 THEN 1800
1820 POKE 53279,0
1830 GOSUB 1900
1840 CX=CX-(ST=11)+(ST=7):CY=CY-(ST=1
     4)+(ST=13)
```

```
1850 IF CX<0 THEN CX=31:CY=CY-1
1860 IF CX>31 THEN CX=0:CY=CY+1
1870 IF CY<0 THEN CY=3
1880 IF CY>3 THEN CY=0
1890 GOTO 1770
1900 POKE SD+CX+CY*40+4,C
1910 POKE ASD+CX+CY*40+4,C
1920 RETURN
5000 REM PRINT DATA
5015 TRAP 1260:OPEN #1,8,0,"P:":PRINT
      #1;"{10 SPACES}---(";C;")-----"
5020 FOR I=0 TO 7:PRINT #1;"
     {10 SPACES}";
5030 A=PEEK(CHSET+C*8+I)
5040 P=128:D=A
5050 FOR J=1 TO 8
5060 IF INT(D/P)=1 THEN PUT #1,88:D=D
     -P:GOTO 5080
5070 PUT #1,32
5080 P=P/2:NEXT J:PRINT #1;"  ";A
5090 NEXT I:PRINT #1;"{10 SPACES}-----
     ----"
5100 CLOSE #1:POKE 54286,192:TRAP 400
     00:GOTO 520
```

Program 2.

```
1000 REM CHLOAD-CHARACTER SET LOADER
1005 OPEN #1,4,0,"D:FONT":REM YOUR FI
     LENAME HERE
1010 X=16:CHSET=(PEEK(106)-8)*256:POK
     E 756,CHSET/256
1020 ICCOM=834:ICBADR=836:ICBLEN=840
1030 POKE ICBADR+X+1,CHSET/256:POKE I
     CBADR+X,0
1040 POKE ICBLEN+X+1,4:POKE ICBLEN+X,
     0
1050 POKE ICCOM+X,7:A=USR(ADR("hhh*LV
     E"),X):REM CALL CIO
1060 CLOSE #1
```

Program 3.

```
100 REM CHPRINT--CHARACTER SET PRINTO
    UT
110 TRAP 340
120 OPEN #1,4,0,"D:FONT":REM YOUR FIL
    ENAME HERE
130 OPEN #2,8,0,"P:":REM CHANGE TO "E
    :" FOR SCREEN
140 PRINT "1 HEX OR 2 DECIMAL";:INPUT
     TYPE
150 DIM HEX$(16),F$(3)
160 HEX$="0123456789ABCDEF"
165 LSB=-1
170 FOR I=0 TO 1023 STEP 8
180 F$="{3 SPACES}":C=INT(I/8)
190 IF TYPE=2 THEN F$(1,LEN(STR$(C)))
    =STR$(C):PRINT #2;F$;":";:GOTO 25
    0
200 LSB=LSB+1:IF LSB=256 THEN LSB=0:M
    SB=MSB+1
210 PRINT #2;"$";HEX$(MSB+1,MSB+1);
230 HINYB=INT(LSB/16):LONYB=LSB-16*HI
    NYB
240 PRINT #2;HEX$(HINYB+1,HINYB+1);HE
    X$(LONYB+1,LONYB+1);": ";
250 FOR J=0 TO 7
260 GET #1,A
270 F$="{3 SPACES}":IF TYPE=2 THEN F$
    (1,LEN(STR$(A)))=STR$(A):PRINT #2
    ;" ";F$;:GOTO 310
290 HINYB=INT(A/16):LONYB=A-16*HINYB
300 PRINT #2;HEX$(HINYB+1,HINYB+1);HE
    X$(LONYB+1,LONYB+1);" ";
310 NEXT J
320 PRINT #2
330 NEXT I
340 CLOSE #1:CLOSE #2
```

Program 4.

```
100 REM CHSET DATAMAKER
102 GRAPHICS 1+16:CHSET=(PEEK(106)-8)
    *256
105 DIM F$(14),OF$(14)
110 POSITION 3,0:? #6;"character set"
120 POSITION 5,2:? #6;"datamaker"
130 ? #6:? #6;"THIS UTILITY CREATES"
140 ? #6;"A SET OF DATA STATE-";
150 ? #6;"MENTS FROM A SAVED"
160 ? #6;"CHARACTER SET.  IT"
170 ? #6;"OPTIMIZES BY ONLY"
180 ? #6;"LISTING CHARACTERS"
190 ? #6;"NOT PRESENT IN THE"
200 ? #6;"STANDARD CHARACTER"
210 ? #6;"SET."
220 ? #6:? #6;"PRESS [redacted]"
230 IF PEEK(53279)<>3 THEN 230
240 GRAPHICS 1+16
250 ? #6;"THE DATA STATEMENTS"
260 ? #6;"WILL BE WRITTEN"
270 ? #6;"AS A list FILE."
280 ? #6;"USE enter TO MERGE"
290 ? #6;"THE DATA WITH YOUR"
300 ? #6;"PROGRAM.":? #6:? #6;"[redacted]
    [redacted]":? #6;"[redacted]"
305 POKE 82,0:POKE 87,0
310 ? "{UP}{DEL LINE}";:INPUT F$:IF F
    $="" THEN 310
315 IF F$="C" OR F$="C:" THEN CASS=1:
    GOTO 332
320 ? "{6 UP}{6 DEL LINE}[redacted]
    {8 SPACES}[redacted]":? :?
330 ? "{UP}{DEL LINE}";:INPUT OF$:IF
    OF$="" THEN 330
335 ? "{3 UP}{3 DEL LINE}[redacted]
    [redacted]{5 SPACES}[redacted]":?
    :?
340 INPUT SLINE
345 CLOSE #1
350 GRAPHICS 2+16:SETCOLOR 4,3,0
360 IF CASS THEN ? #6:? #6;"POSITION
    CHARACTER":? #6;"SET TAPE,HIT [redacted]
    [redacted]"
```

```
370 POSITION 5,6:? #6;"working{3 N}"
375 GOSUB 1000:REM LOAD CHARACTER SET
377 IF CASS THEN ? #6;"{CLEAR}INSERT
    OUTPUT TAPE,":? #6;"PRESS RETURN"
380 OPEN #2,8,0,OF$:POSITION 5,6:? #6
    ;"WORKING{3 N}"
381 ? #2;SLINE;"CHSET=(PEEK(106)-8)*2
    56:FOR I=0 TO 1023:POKE CHSET+I,P
    EEK(57344+I):NEXT I"
382 ? #2;SLINE+1;"RESTORE ";SLINE+5
383 ? #2;SLINE+2;"READ A:IF A=-1 THEN
     RETURN"
384 ? #2;SLINE+3;"FOR J=0 TO 7:READ B
    :POKE CHSET+A*8+J,B:NEXT J"
385 ? #2;SLINE+4;"GOTO ";SLINE+2
387 LINE=SLINE+4
390 FOR I=0 TO 127:F=0
400 FOR J=0 TO 7
420 IF PEEK(CHSET+I*8+J)<>PEEK(57344+
    I*8+J) THEN F=1
430 NEXT J
440 IF  NOT F THEN 460
445 LINE=LINE+1
450 ? #2;LINE;" DATA ";:? #2;I;:FOR J
    =0 TO 7:? #2;",";PEEK(CHSET+I*8+J
    );:NEXT J:? #2
460 NEXT I:? #2;LINE+1;"DATA -1"
470 ? "All finished!  Use ENTER ";CHR
    $(34);OF$
480 ? "to merge the file."
490 END
1000 REM HIGH-SPEED LOAD OF CHARACTER
      SET
1005 OPEN #1,4,0,F$:REM OPEN FILE
1010 X=16:REM $10
1020 ICCOM=834:ICBADR=836:ICBLEN=840
1030 POKE ICBADR+X+1,CHSET/256:POKE I
     CBADR+X,0
1040 POKE ICBLEN+X+1,4:POKE ICBLEN+X,
     0
1050 POKE ICCOM+X,7:A=USR(ADR("hhh{3}LV
     {2}"),X):REM CALL CIO
1060 CLOSE #1:RETURN
```

Character Set Utilities

Fred Pinho

In addition to providing some useful utilities for working with redefined characters, this article discusses memory allocation and the various ways of storing machine language subroutines.

The Atari computer has the ability to redefine its character set at will. Making full use of this power, however, requires some programming. It isn't available at the flick of a switch or the touch of a key. To help in this effort, I'd like to present two simple machine language utilities for use in the text modes (GRAPHICS 0, 1, 2).

To set the stage, here's a brief overview of the Atari character set. A character set is really just a table of shapes which defines what a character will look like when printed to the screen or to a printer. The characters are used solely for communication with the human operator. The computer, as always, is manipulating numbers, not letters or graphics symbols. However, we think very clumsily, if at all, in pure numbers. The computer graciously converts its thoughts into symbols that we can understand and manipulate.

The set is stored in Read Only Memory (ROM) beginning at memory location 57344. Each character is stored within eight bytes of information. This provides an eight-by-eight grid for defining a character. Within each byte, a one means a dot is turned on by the video display. A zero leaves the dot off. There are 128 regular characters plus inverse. Since inverse characters can be generated from the normal, only the regular characters are stored. Thus, the Atari character set contains 1024 bytes (8*128).

Before it can do its thing, the computer needs to know the location of the character set. This is stored in two registers: the "shadow" register (CHBAS=756 decimal) and the hardware register (CHBASE=54281). The computer actually uses the hardware register to locate the character set. However, every 60th of a second, during vertical blank of the TV screen, it goes to the shadow register and transfers its contents to the hardware register.

From BASIC, if you wish to change the location of the character set, you must POKE the new location into the shadow register, not the hardware register. If you POKEd into the hardware register, you would see nothing, as it would be wiped out immediately by the copy from its shadow. The setup of the two registers makes it impossible to have multiple character sets on the screen at the same time when using BASIC. However, it can be done with machine language subroutines.

What do you store in the character registers? You store the page number of the beginning memory location of the set. What's a page number? The computer breaks down memory into 256-byte "pages" (recall that the range of numbers that can be stored in a single byte is 0 to 255 for a total of 256). The page number is just a fancy way of indicating a multiple of 256. To get the page number, divide your memory address by 256. If your answer doesn't come out to an exact number, you're in trouble. Run, don't walk, to another memory location that is exactly divisible. This is important because your programs will not work if your page calculation is incorrect.

The full character set, plus inverse, can be displayed in graphics mode 0. Inverse characters cannot be displayed in graphics modes 1 and 2. In addition, these modes can display only half of the full set at any one time (64 characters). Thus, in these modes, you are limited to displaying either capital letters, numbers and punctuation, or lowercase letters and graphics symbols. From BASIC, you can't have both.

As you can see, the power of the redefinable character set is there, but it's not available without some programming effort. What's required to use redefined and multiple character sets? Three main steps must be considered.

1) Relocating the original character set from ROM into Random Access Memory (RAM).

2) Revising the relocated, RAM-based, character set.

3) Providing the computer with a program that can switch between character sets at predictable times during the TV display process.

Relocating The Character Set

Relocating the character set is simple in principle: PEEK each ROM location and POKE it into the desired RAM location. The first problem arises as to where to store the set. One common solution is to lower RAMTOP. This memory location (106) de-

fines the upper limit of available memory. By POKEing a lower page number into this location, you can fool the computer into thinking it has less memory than actually installed. The character set can then be relocated to this area. Note, however, that this hidden area is not completely safe. Certain programming operations can cause unanticipated visits into this area by the computer. This would have a disastrous effect on your characters. Solutions to this problem would involve avoiding the guilty program commands and/or allocating extra, wasted memory above RAMTOP.

Another way is to relocate the set just below the display list. You have to be careful not to overrun the display list. Also, you have to plan your program so that it doesn't expand into your new character set. Even if your program is properly sized, you can still run into problems with an overly-obese run/time stack. This software stack is established by BASIC and resides just above your BASIC program area. It stores needed information for GOSUB and FOR/NEXT routines. If you exit from a FOR/NEXT loop before it finishes counting down, an entry is left on the stack. If this loop is used frequently, the stack will grow until it attacks your character set. The solution is careful use of the POP command to clean up the stack whenever you prematurely exit from a loop. Table 1 will allow you to relocate your set without interfering with the display list.

Another possibility would be to relocate your character set into a string. Just DIMension it for the proper size and then use the ADR function to locate the first memory location. This method would certainly provide a safe location. However, another problem arises.

Note that when you relocate a character set there are certain limitations. The full set (GRAPHICS 0) must start on a 1K boundary (i.e., the first memory location must be a multiple of 1024). The reduced set for GRAPHICS 1 and 2 must start on a 1/2K boundary (multiple of 512). This poses no problem for the first two storage methods. However, if you wish to use the string-storage method, you will have to expend some extra programming effort. You must insure that the string begins on the proper memory boundary.

Which storage method is best? I'll leave that up to you and to the demands of your program.

Relocation of the Character Set Beneath the Display List.

GRAPHICS MODE	Relocate at the Indicated Offset (in Pages) from RAMTOP	
	Full Set (1024 bytes)	Half Set (512 bytes)
0	8	Won't work
1	8	6
2	8	4
3	8	4
4	8	6
5	12	8
6	16	12
7	24	20
8	36	34

Note: Graphics modes are included since certain applications may require plotting of characters to the graphics screen.

Example:

Relocate the first half of the character set so it resides just below the GRAPHICS 1 display list. Label the first memory location of the set as BEGIN.

BEGIN = (PEEK(106)-6)*256

Once you've chosen the location, the next step is to move the character set. Using the PEEK-POKE method works OK, but it's too slow. If you've chosen to store your new character set in a string, then you could also relocate your set using the Atari's string handling routines, which are fast. This involves modifying the variable value table to fool the computer into thinking that one string is located at the ROM-based character set. This technique is described by Bill Wilkinson (**COMPUTE!**, January 1982, #20). Use of this technique would also solve your memory boundary problem. Note that there is a BASIC bug that does not allow the correct movement of string characters in multiples of 256. Thus you would have to transfer either 513 or 1025 bytes, instead of 512 or 1024 bytes. The most general, quickest and hassle-free method is to use a machine language subroutine. As usual, a decision must first be made. Where do you store your routine in order to protect it from BASIC's vora-

cious appetite? Here are three good methods:

1. **In page six (begins at memory location 1536).** This page of memory was set aside by the Atari designers for use by the BASIC programmer. Generally, you can safely store machine language programs here. You then access them by the USR command (X=USR (1536)). Note, however, that the first 128 bytes of page six are not always safe. If you perform cassette input/output during the program, you could lose this first block of memory. To be absolutely safe, store your routine only in the last half of the page.

2. **As a string.** The method that I like is to store the program commands as graphics symbols within a string. Take the machine language number (in decimal), go to the ATASCII table (Appendix C in your *Atari BASIC Reference Manual*), find the equivalent graphics symbol, and type it into the string. You then access the machine language program by:

```
X=USR (ADR(MVCHR$))
```

Remember to DIMension this string first. With this method you are not limited in the size of your program as you would be with page six storage, and your program is *safe.*

3. **Within an array.** Atari BASIC allocates memory within the string/array area in the order in which you DIMension it. This location doesn't change thereafter. Note that this is not true of many other machines. Thus, if you DIMension a string followed by an array, you can locate the array relative to the string by use of the ADR function.

```
10 DIM AA$(1), MVCHR(32)
20 X=USR(ADR(AA$)+1)
```

AA$ is meaningless except that you can determine its memory location. If you POKE your machine language program into the array MVCHR, you can always access it in the memory location following AA$. This method, however, can chew up large gobs of memory. Each cell in the array takes six bytes. Be careful to type these DIMension statements sequentially.

Which method should you use? Again, it depends. I like the string method since it is safe and memory-efficient. The efficiency arises from storing data with a single symbol, rather than storing each integer of the number plus a comma in DATA statements. You also avoid the overhead of the program lines required for READing the data statements.

To aid your programming efforts, I've included a machine language routine in Programs 1, 2, 3, 4, and 5. This routine will rapidly relocate your character set. Program 1 is listed in assembly language. The others are BASIC versions which demonstrate the various ways of storing a machine language routine in memory. This routine uses four zero-page memory locations (203-206). These locations have been set aside by Atari for programming use and are safe. To use them, POKE the address of your new character set into locations 203 and 204. As always, the least significant byte of the 16-bit address is POKEd first, followed by the high byte. Since the set must start on either a 512 or 1024-byte boundary, the least significant byte must always be zero. Then POKE the page number of the character set address into location 204. Memory locations 205 and 206 are set by the machine language routine to point to the character set in ROM. Note that location 205 is also set to zero. Finally, do a USR call to your routine.

Switching Between Character Sets

Remember that, because of the setup of the two character set registers, it's not possible to use more than one set at a time via BASIC. You can display as many sets as you wish, however, with display list interrupts. Well, almost. Actually you're limited to some extent by memory availability and the constraints of the display list. I won't go into detail here; this subject has been covered by numerous articles. It's enough to say that:

1) The interrupt will cause the 6502 processor to stop at a given scan line of the TV display.

2) It will then execute a machine language routine of your choice. The address of the routine must be specified in locations 512 and 513.

3) Once done, the 6502 will then merrily resume its TV display.

If the interrupt is done properly, all the action will occur while the TV beam is off the visible part of the screen. Thus the changes performed will appear instantaneous. Program 3 is a routine to allow the use of a redefined character set in the upper window of a GRAPHICS 1 or 2 display and the standard set in the text window. This routine simply loads the page number of the old ROM-based character set into the hardware character base register (CHBASE = 54281). What good is that, you ask?

This is done normally anyhow. Here's the strategy:

1) In BASIC, load the address of your new set into the shadow character base register. This is copied into the hardware register at the start of each TV screen display (every 60th of a second).

2) Set a display list interrupt at the last line of the GRAPHICS 1 or 2 screen. Then the interrupt will begin at the first line of the text window.

3) The interrupt routine loads the address of the standard character set into the hardware register. Thus the text window regains all the standard characters.

4) Things remain standard until vertical blank. At this time the TV beam is brought back to the top of the screen in order to begin a fresh sweep of the display. During vertical blank the contents of the shadow are automatically copied into the hardware register. Thus we've restored the new character set.

Note that the interrupt routine stores a number (any number will do) into memory location WSYNC (54282). This causes the processor to wait until the end of the blank period at the start of the next horizontal line. Thus the character set is switched while the electron beam is off the visible portion of the display. The result is a neat, clean change. Program 6 is an assembly listing of the display list interrupt routine, with decimal equivalents in parentheses to use in your DATA statements. (See line 31000 of Program 7.)

Notes: To set the interrupt, do the following:

1. *Load the routine into memory.*
2. *POKE the address for the interrupt routine into locations 512 and 513. POKE the low byte of the two-byte address first.*
3. *POKE the interrupt into the display list. Set the line before the one you want.*
4. *Last, enable the interrupt by POKEing 192 into location 54286. To disable the interrupt, POKE zero into this location.*

Pulling It Together

Program 7 demonstrates the techniques discussed. It prints an identical set of characters to the text window and to the GRAPHICS 1 screen. The expected characters are seen in the text window. However, the GRAPHICS 1 display shows a band of archers beseiging a castle.

Line No.	
1	Calls initializing subroutine and turns screen display back on.
10-40	Draws a border using a redefined character.
50-60	Prints characters.
100	Uses STOP instead of END, as this allows you to experiment and print to the GRAPHICS 1 screen in direct mode. END does not.
29000	Turns off display to speed processing. Calculates location of new character set (BEGIN) at six pages below RAMTOP.
29010	POKEs the low and high bytes of the new set location into memory locations 203 and 204.
29020	DIMensions the string for relocation of the old set. Defines the string and calls the machine language routine.
29030-29040	Reads the redefined character data and POKEs it into the relocated set.
29050-29060	Calculates the location of the display list. POKEs an interrupt into the last GRAPHICS 1 line. Reads the interrupt routine into page six of memory. POKEs the address of page six into locations 512 and 513. Finally, it enables the interrupt.
30000-30060	New character data.
31000	Data for interrupt routine.

You can do a simple experiment with this routine. To prove the necessity of writing to WSYNC in the interrupt routine, eliminate the fourth through sixth numbers in line 31000 (i.e., 141, 10, 212). Also change line 29050 to FROM X=0 TO 7. Run the program again. You'll see that the last scan line of the GRAPHICS 1 screen stops about halfway across the screen. Also, the point at which it stops tends to jiggle annoyingly. Other weird lines appear when you hit a key. Finally, delete POKE 756, BEGIN/256 from line 29000. RUN and you'll see only normal characters.

There you have it. These programs only scratch the surface. With some further programming, you can have even more character sets on the screen simultaneously. You can also make your characters blink or change color without interfering with your BASIC program.

Program 1. Character Set Relocater – Assembly Listing.

```
      PLA            ;Pull unused byte
                      off stack
      LDA #0
      STA $CD        ;Low byte of
                      ROM-based
                      character set
      TAY            ;Set register
                      Y to zero
      LDA *          ;*See Notes
      STA $CF        ;High byte of ROM set
LOAD  LDA ($CD),Y    ;Load from ROM
      STA ($CB),Y    ;Store into RAM
      INY            ;Increment Y
      BNE LOAD       ;Loop 256 times
      INC $CC        ;Increment RAM page
                      number
      INC $CE        ;Same for ROM
      LDA $CE
      CMP *          ;Compare to final
                      page number
                      *See Notes
      BNE LOAD       ;If not done,
                      loop again
      RTS            ;Return
```

Notes on Programs 1, 2, 3, 4, and 5:

Programs 2 through 5 are BASIC versions of Program 1. To customize, make the following changes (if needed). Remember to convert the DATA numbers below to character equivalents when using Program 3.

1. To relocate the entire character set, be sure the 8th DATA number is 224, and the 25th DATA number is 228.

2. To relocate only the first half of the character set (uppercase, numbers, punctuation), be sure the 8th DATA number is 224, and the 25th DATA number is 226.

3. To relocate only the second half of the character set (lowercase, graphics symbols), be sure the 8th DATA number is 226, and the 25th DATA number is 228.

Program 2. Store machine language routine in page six.

```
10 RESTORE 30:FOR X=0 TO 27:READ Y:PO
   KE 1536+X,Y:NEXT X
20 Z=USR(1536)
30 DATA 104,169,0,133,205,168,169,224
   ,133,206,177,205,145,203,200,208,2
   49,230,204,230,206,165,206,201,226
40 DATA 208,239,96
```

Program 3. Store machine routine as a string.

```
10 DIM MVCHR$(28)
20 MVCHR$="h[illegible]{,}{[illegible]}{[illegible]}{[illegible]}{[illegible]}{[illegible]
   [illegible]"
30 Z=USR(ADR(MVCHR$))
```

Program 4. Store machine language routine as a string converted from DATA statements.

```
10 DIM MVCHR$(28)
20 RESTORE 40:FOR I=1 TO 28:READ Y:MV
   CHR$(I)=CHR$(Y):NEXT I
30 Z=USR(ADR(MVCHR$))
40 DATA 104,169,0,133,205,168,169,224
   ,133,206,177,205,145,203,200,208,2
   49,230,204,230,206,165,206,201,226
50 DATA 208,239,96
```

Program 5. Store machine language routine within a matrix.

```
10 DIM AA$(1),MVCHR(27)
20 RESTORE 40:FOR X=1 TO 28:READ Y:PO
   KE ADR(AA$)+X,Y:NEXT X
30 Z=USR(ADR(AA$)+1)
40 DATA 104,169,0,133,205,168,169,224
   ,133,206,177,205,145,203,200,208,2
   49,230,204,230,206,165,206,201,226
50 DATA 208,239,96
```

Program 6. Display List Interrupt.

```
PHA                 ;Save accumulator
                     (decimal 72).
LDA # $E0           ;Load ROM page num-
                     ber (decimal 169,
                     224).
STA WSYNC           ;Write to WSYNC
                     (decimal 141,10,
                     212).
STA $D409           ;Store page number
                     in hardware regi-
                     ster (decimal 141,
                     9,212).
PLA                 ;Restore accumu-
                     lator (decimal
                     104).
RTI                 ;Return (decimal
                     64).
```

Program 7. Demonstration.

```
1 GOSUB 29000:POKE 559,34
10 FOR X=0 TO 18 STEP 2:COLOR 154:PLO
   T X,0:COLOR 26:PLOT X,19:NEXT X
20 FOR X=1 TO 19 STEP 2:COLOR 26:PLOT
    X,0:COLOR 154:PLOT X,19:NEXT X
30 FOR Y=2 TO 18 STEP 2:COLOR 154:PLO
   T 0,Y:COLOR 26:PLOT 19,Y:NEXT Y
40 FOR Y=1 TO 17 STEP 2:COLOR 26:PLOT
    0,Y:COLOR 154:PLOT 19,Y:NEXT Y
50 POSITION 4,6:? #6;"!%&&%#":POSITIO
   N 12,6:? #6;"---"
60 ? "!%&&%#";:? "    ";:? "---"
100 STOP
29000 GRAPHICS 1:POKE 559,0:BEGIN=(PE
      EK(106)-6)*256:POKE 756,BEGIN/2
      56
29010 SHI=BEGIN/256:SLO=0:POKE 203,SL
      O:POKE 204,SHI
```

```
29020 DIM MVCHR$(28):MVCHR$="h[illegible](,}
      {[illegible]}{[illegible]}{[illegible]}{[illegible]}{[illegible]}
      [illegible]":Z=USR(ADR(MVCHR$)):
      RESTORE 30000
29030 READ X:IF X=-1 THEN 29050
29040 FOR Y=0 TO 7:READ Z:POKE X+Y+BE
      GIN,Z:NEXT Y:GOTO 29030
29050 DLST=PEEK(560)+256*PEEK(561):PO
      KE DLST+24,134:RESTORE 31000:FO
      R X=0 TO 10:READ Y
29060 POKE 1536+X,Y:NEXT X:POKE 512,0
      :POKE 513,6:POKE 54286,192:RETU
      RN
30000 DATA 8,165,231,231,231,255,219,
      255,231
30010 DATA 24,161,162,228,232,240,227
      ,227,227
30020 DATA 40,24,60,126,66,126,219,25
      5,255
30030 DATA 48,0,0,0,36,36,36,255,255
30040 DATA 104,0,44,68,254,76,44,20,2
      2
30050 DATA 208,255,255,255,255,255,25
      5,255,255
30060 DATA -1
31000 DATA 72,169,224,141,10,212,141,
      9,212,104,64
```

Notes on Program 7:

1. To set interrupts for the text window of each of the text modes, change line 29050 as follows:

Text Mode	Change to
1	POKE DLST + 24, 134
0	POKE DLST + 24, 130
2	POKE DLST + 14, 135

2. To add a text window to GRAPHICS 0, POKE 4 into memory location 703. See "Add a Text Window to GRAPHICS 0," reprinted in this book.

3. If you change the mode as in 2 above, don't forget to adjust the character set storage (BEGIN) in line 29000. Also remember the GRAPHICS call in line 29000.

Chapter 4

Animation With Character Graphics

TextPlot

Charles Brannon

TextPlot is a machine language graphics utility that lets you mix text and graphics. It is designed to work with the four-color graphics modes three, five, and seven. It will place any ATASCII character – upper/lowercase, graphics, numbers, and special symbols in normal or reverse field – on the graphics screen in any of three colors. The size of the characters varies in proportion to the pixel size: GRAPHICS 3 characters are *four times* as large as those in GRAPHICS 7, whose characters are the same size and proportions as those in GRAPHICS 2 (text mode). Through standard display list modification, any of the three sizes of text can be mixed with the other modes. TextPlot enables you to use a total of *eight* text modes. (See the description of the "bonus" text modes later.)

Text On Graphics Lines

TextPlot, unlike the text modes, can be mixed on the same line with normal graphics. You can label charts and graphs, or quickly draw pictures with the graphics characters and then embellish them. TextPlot even works with an alternate character set, so you can design special "shapes" and move them around the screen for high-speed animation. The text in graphics mode three is *huge*, a real eyecatcher. Unlike the other text modes, TextPlot lets you position any character at any possible vertical resolution (although horizontally it's the same). And all this was without modifying the display list!

Luckily, TextPlot is easy to use. You load it into memory (it goes into the reserved memory at $600 hex) with a BASIC loader or BINARY LOAD, via DOS. You then select the graphics mode in which to use it with the ordinary GRAPHICS command. (TextPlot works in either full-screen or window modes.) You then "plot" each character with the command:

```
A=USR(1536,chr,color,column,row)
```

Don't let this machine language call intimidate you. It merely enables a USeR command. The other variables for the function communicate with TextPlot. If you leave one out, or add an

extra one, TextPlot will ring the bell to warn you.

CHR: The ASCII value of the desired character [like ASC("K")].

COLOR: The color of the character (just like the COLOR statement, 1-3).

COLUMN: The horizontal position of the character. This depends on the mode:

Mode	Max Columns	Max Rows
3	5	16
5	10	40
6	10	88
7	20	88
8	20	184

ROW: The vertical position of the character. This also depends on the mode (see above), and is the line at which you want the character to start. Remember that each character is just eight lines of dots, so they can start at any pixel position vertically. The horizontal resolution is limited by the internal storage of graphics information on the screen.

So, to place a blue capital letter "A" on the screen in graphics mode three, at the second column and tenth row, use the command:

```
A=USR(1536,65,3,2,10)
```

where 65 is the ATASCII value of "A"; 3 is the color; 2 is the column; 10 is the row. Strings of text can be placed on the screen as well:

```
DIM T$(20)
T$="That's Incredible!"
GRAPHICS 7+16
FOR I=1 TO LEN(T$)
  A=ASC(T$(I,I))
  V=USR(1536,A,1,I,2)
NEXT I
```

Notice that you can use any variable with the USR function, not just A.

Bonus Text Modes

TextPlot was designed for the four-color graphics modes. Strange things can happen if you use it in any other mode. In modes six and eight, however, you will indeed get text. In GRAPHICS 6, the characters are the same size as those in

GRAPHICS 5. There is a blank line between each row of dots in each character. A character plotted in COLOR 1 or COLOR 2 will also skip horizontally. COLOR 3 will create characters divided into "bands." The effect is similar to the IBM logo (see Figure 1). This same oddity results in "artifacting" in GRAPHICS 8. What does that mean? You get three colors of text in GRAPHICS 8! Depending on background and dot colors, COLOR 1 is purple, COLOR 2 is green, and COLOR 3 is white. (See Chapter 6 for more information on artifacting.) The text is twice the width of GRAPHICS 0 characters, but the same height, just like GRAPHICS 1. Other strange patterns and effects can be generated by using numbers other than 0-3 in the color assignment. A seven creates a "3-D" overlay effect, for example.

I have included a sample program that lets you type on the screen using a flashing cursor. It works in graphics mode seven. You can use all the standard keys, but only a few of the editing keys work. What can I say? It's not supposed to be a word processor! The lines from 20000 and up will place TextPlot into memory at page six. You can save them to disk or tape and merge them with other programs using the LIST/ENTER combination.

For Cassette	For Disk
Rewind cassette, press PLAY & RECORD, and enter:	Enter:
LIST"C:",20000,32767	LIST"D:TXTPLT.ENT", 20000,32767
Press RETURN twice.	and press RETURN.
To merge with a program already in memory:	
Rewind tape, press PLAY, enter:	Enter:
ENTER "C:"	ENTER "D:TXTPLT.ENT"
and press RETURN twice.	and press RETURN.

Advanced readers may want to know how TextPlot works (if you haven't figured it out already). You are probably familiar with how to plot characters on the GRAPHICS 8 screen by PEEKing the character generator and then placing these bit patterns directly into the screen memory for GRAPHICS 8. It works because each byte in GRAPHICS 8 (and modes four and six, too) displays eight dots, or pixels. A one-bit in the byte means a "lit" pixel and a zero is a dark ("background") dot. The four-

color modes have to split the load between two bytes. Each byte displays four pixels. Two bits hold the color (binary): 01 color one, 10 color two, 11 color three. (See Figure 2.) TextPlot uses the character generator (indirectly through CHBAS, 756 decimal) to get the bit map and then "pulls" the byte accordion-style into two color bytes. Theoretically, any character could be a mixture of the three colors, but it's hard to implement and use. (Unless you use Antic Display modes 4 or 5....)

Using TextPlot As A BINARY FILE

The Atari DOS lets you store machine language files on the disk and load them back, both by DOS menu selections. You can even have TextPlot load in automatically with the DOS, if you're sure you'll always need it. After placing TextPlot into RAM, go to DOS with the command: DOS. If you have DOS 2.0S, there will be a pause as the Disk Utility Package loads. The DOS menu should be displayed. Type K ‹RETURN›. After the prompt, enter:

TXTPLT.OBJ,0600,06FF ‹RETURN›

If you want TextPlot to automatically load with DOS, enter:

AUTORUN.SYS,0600,06FF ‹RETURN›

instead. If you don't do this, you'll have to go to DOS and enter L (Load) and reply with TXTPLT.OBJ to load it and B ‹RETURN› to exit to BASIC.

Figure 1.

GRAPHICS 6

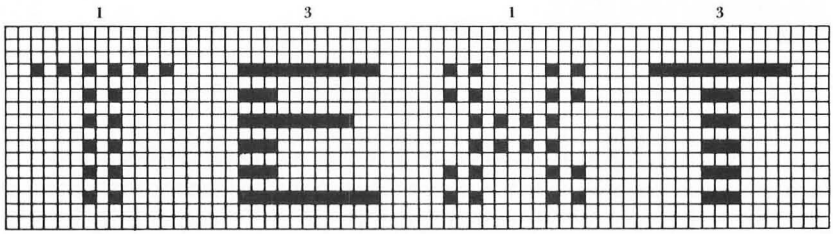
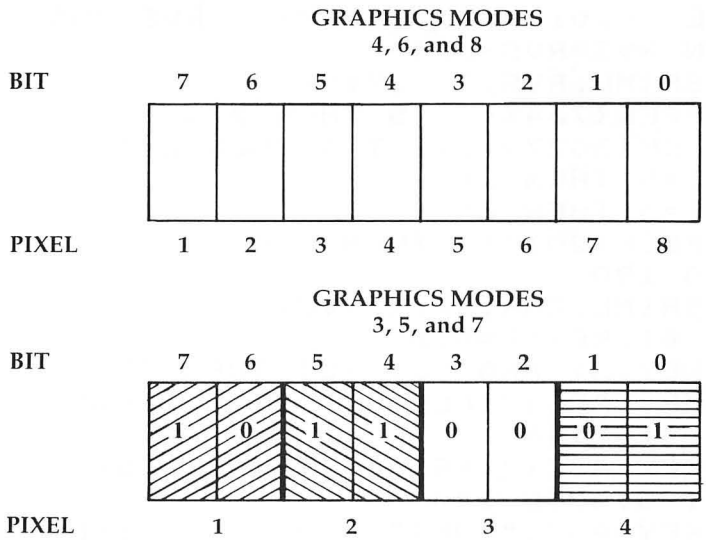


Figure 2.



Program.

```
10 REM SUPER SCREEN-TextPlot Demo
20 REM Use all the ATARI characters
30 REM including cursor up/down
40 REM left/right, backspace, RETURN,
50 REM etc.  Press CAPS/LOWR to
60 REM select upper or lower case,
70 REM as usual.  Atari Logo key
80 REM toggles reverse field.
90 REM Press console buttons for diff
   erent colors
100 REM ESC switches modes (7 vs. 8)
110 ML=1536
120 IF PEEK(ML)=0 THEN GOSUB 470
130 XL=19:YL=11:DIM CHAR$(480),C$(480
    )
140 CHAR$=" ":CHAR$(480)=" ":CHAR$(2)
    =CHAR$:C$=CHAR$
150 GRAPHICS 7+G+16:OPEN #1,4,0,"K:"
160 IF G=1 THEN SETCOLOR 2,0,0
170 LM=1:X=LM:Y=0:C=1
```

```
180 POS=X+Y*20+1:CHR=ASC(CHAR$(POS,PO
    S)):RVS=CHR:SC=ASC(C$(POS))-31
190 POKE 20,0:RVS=RVS+128:IF RVS>255
    THEN RVS=RVS-256
200 A=USR(ML,RVS,C,X,Y*8)
210 IF PEEK(764)<>255 THEN 270
220 T=PEEK(53279):IF T=6 THEN C=1
230 IF T=5 THEN C=2
240 IF T=3 THEN C=3
250 IF PEEK(20)<15 THEN 210
260 GOTO 190
270 A=USR(ML,CHR,SC,X,Y*8)
280 GET #1,KEY:E=0:DL=E
290 IF KEY>31 AND KEY<123 THEN 430
300 IF KEY=ASC("{CLEAR}") THEN CLOSE
    #1:GOTO 140
310 IF KEY=ASC("{ESC}") THEN CLOSE #1
    :G=1-G:GOTO 140
320 IF KEY=ASC("{UP}") THEN Y=Y-1:E=1
330 IF KEY=ASC("{DOWN}") THEN Y=Y+1:E
    =1
340 IF KEY=ASC("{LEFT}") THEN X=X-1:E
    =1
350 IF KEY=ASC("{RIGHT}") THEN X=X+1:
    E=1
360 IF KEY=155 THEN X=LM:Y=Y+1:E=1
370 IF KEY=ASC("{BACK S}") THEN X=X-1
    :KEY=32:DL=1
380 IF X<LM THEN X=XL:Y=Y-1
390 IF X>XL THEN X=LM:Y=Y+1
400 IF Y>YL+YL*G THEN Y=0
410 IF Y<0 THEN Y=YL
420 IF E THEN 180
430 A=USR(ML,KEY,C,X,Y*8)
440 POS=X+Y*20+1:CHR$(POS,POS)=CHR$(
    KEY):C$(POS,POS)=CHR$(31+C)
450 IF DL=0 THEN X=X+1:IF X>XL THEN X
    =LM:Y=Y+1:IF Y>YL THEN Y=0
460 GOTO 180
470 ML=1536:FOR I=0 TO 252:READ A:POK
    E ML+I,A:NEXT I:RETURN
480 DATA 104,240,10,201,4,240
```

```
490 DATA 11,170,104,104,202,208
500 DATA 251,169,253,76,164,246
510 DATA 104,133,195,104,201,128
520 DATA 144,4,41,127,198,195
530 DATA 170,141,250,6,224,96
540 DATA 176,15,169,64,224,32
550 DATA 144,2,169,224,24,109
560 DATA 250,6,141,250,6,104
570 DATA 104,141,251,6,104,104
580 DATA 141,252,6,14,252,6
590 DATA 104,104,141,253,6,133
600 DATA 186,166,87,169,10,224
610 DATA 3,240,8,169,20,224
620 DATA 5,240,2,169,40,133
630 DATA 207,133,187,165,88,133
640 DATA 203,165,89,133,204,32
650 DATA 228,6,24,173,252,6
660 DATA 101,203,133,203,144,2
670 DATA 230,204,24,165,203,101
680 DATA 212,133,203,165,204,101
690 DATA 213,133,204,173,250,6
700 DATA 133,187,169,8,133,186
710 DATA 32,228,6,165,212,133
720 DATA 205,173,244,2,101,213
730 DATA 133,206,160,0,162,8
740 DATA 169,0,133,208,133,209
750 DATA 177,205,69,195,72,104
760 DATA 10,72,144,8,24,173
770 DATA 251,6,5,208,133,208
780 DATA 224,1,240,8,6,208
790 DATA 38,209,6,208,38,209
800 DATA 202,208,228,104,152,72
810 DATA 160,0,165,209,145,203
820 DATA 200,165,208,145,203,104
830 DATA 168,24,165,203,101,207
840 DATA 133,203,144,2,230,204
850 DATA 200,192,8,208,183,96
860 DATA 169,0,133,212,162,8
870 DATA 70,186,144,3,24,101
880 DATA 187,106,102,212,202,208
890 DATA 243,133,213,96,0,1
900 DATA 28
```

Using TextPlot For Animated Games

David Plotkin

When typing in this program, be especially careful typing the numbers in the DATA statements and USR commands. A mistake could cause your machine to lock up, *that is, no longer respond to the external world, requiring a power-on reset.*

If you're like me, the first thing you did when you bought your new Atari was run out to buy some games for it, probably with visions of multicolored, arcade-style entertainment in mind. The computer store where I purchased my Atari also sells Apples. The wide assortment of exciting, machine language games available on the Apple and *not* on the Atari was a real disappointment. Time and time again I saw fascinating games which were not available to me. The recent release of many new Atari programs has somewhat alleviated this, but the problem still exists. To make things even more frustrating, many interesting games are not all that complex from a programming standpoint.

I decided to try my hand at programming these games myself. Having completed the book on how to program in BASIC, I charged ahead and wrote my first "arcade-style" game, which I entitled "Space Rocks." It was a home-grown version of *Asteroids*. The program had it all: graphics, sound, multiple missiles in flight at once, a fancy space ship, scoring, and music. It was also extremely slow. I had spent two weeks on it and each move took almost a minute. Ridiculous? Of course. I tried to speed it up by simplifying the graphics, but never did get it running very fast.

The next step was to try writing a program in a text mode. The Atari can manipulate text quickly, so I had limited success. Using a custom-designed character set also added to the text-mode games. Nevertheless, when there is more than one character to move, it can still be quite slow. I briefly considered learning machine language, but it's not something I'm eager to tackle.

The program "TextPlot" (see previous article) is a first-rate

gaming tool. As the author said, it allows you to use text and text characters in graphics modes. It also works with an alternate character set, as also mentioned briefly. But here's the kicker – since it draws the text character (and erases it also) using a machine language routine, it can be used to animate in high resolution graphics modes at machine language speeds. Thus, your character "A", redefined to a space ship or missile, literally zips across the screen, and five or ten "A's" can move across the screen without the frustrating BASIC characteristic of "taking turns."

By drawing the non-moving portions of your picture in a BASIC graphics mode, and the moving portion using TextPlot, you can write some colorful and challenging games. The program below demonstrates my own efforts in this regard, which I will tell you about shortly. But first some pointers:

1. Animation is done by drawing, erasing, and redrawing in a new position. The erasing can be done in two ways. You can call the USR command with the character ASCII code, but in the *background* color. Or you could call USR command with the ASCII code 32 (blank space) in *any* color. By looping and using a variable either in the color slot or in the ASCII code slot, drawing and erasing is easy. Increment the X and/or Y coordinates (such as MX1 and MY1 in the program) between erase and the redrawing, and the character moves smoothly across the screen. This incrementing, by the way, was done in BASIC (MX1 = MX + 1, etc.) and seems to be the limiting factor in how many characters can move across the screen at once without significant "taking turns."

2. It is possible to define a creature or object which consists of two or three redefined characters which move together. It is best to increment the location of all three characters and then call the machine language routine to move them the most smoothly.

3. There is a large difference between vertical and horizontal resolution. Moving a character one space horizontally is equivalent to moving eight spaces vertically. Remember this when moving diagonally. Also, BASIC commands such as DRAWTO, PLOT, LOCATE, etc., work on the graphics mode coordinate system. Thus, the horizontal location in mode seven can vary from zero to 159, but the X coordinate input to the USR call can vary only from zero to 19, normally. Therefore, X coordinate = horizontal location/eight. The vertical resolution is the same as the Y coordinate.

Note that, in the program, I have varied the X coordinate from 60 to 79 instead of from zero to 19. What this does is move the character down one pixel for each multiple of 20 (60 to 79 moves the character down three pixels from where it would be at zero to 19). A character moving horizontally will pass across the screen lower and lower at higher values of the X coordinate without changing the Y coordinate. This invalidates the relationships shown above between coordinates and screen position, which work only if the X coordinate is between zero and 19.

4. A LOCATE statement meant to find or detect one of the generated text characters cares not what the character is, but only what the color of the character is. This is because the text character is just a series of pixels set to a particular color.

5. The alternate character set is located in an area of RAM protected by POKEing a lower number of pages into location 106, which stores the number of pages (multiply by 256 to get bytes) available in memory. This is a fairly common technique of protecting memory, since the computer doesn't know about the memory above location 106 (see line 3200 in the program) and thus doesn't use it.

In the original version of the character generator, a step-back of five pages (1280 bytes) was used. The character set is four pages (1K) long, plus one extra. This works fine in graphics mode zero, but does not work for this program. I found that the minimum step-back is 16 pages (4K), although any multiple of 4K (32 pages or 48 pages) will work. Intermediate values led to part of the screen being blank or to runny dots and lines being displayed. A final point on this: after every GRAPHICS command, you need to include a POKE 756,PEEK(106)+1 to point the Character Base (CHBAS) address to the redefined character set, since the GRAPHICS command resets the pointer to the ROM character set.

Rules Of The Game

Now to the program. You are chief gunnery officer of the Space Fortress Reliable, located at the outermost fringes of the Galactic Empire. Although the fortress is protected by shields, there are four "channels" through the shields to allow for supply ships and transportation of personnel. Since attacking vessels can also make use of these channels, a big laser is mounted to fire down each of the channels.

The channels are located directly above, below, left, and right of the fortress. Their width is such that only one ship at a time can attack from any direction. The laser is aimed in the appropriate direction by pushing the joystick in that direction. Once the laser is aimed, it fires automatically.

As the attack progresses, however, and energy is used up, the shields begin to withdraw towards the fortress to maintain integrity. The enemy ships can come out of hyperspace and begin the attack through the channels closer to the fortress, so you have less time to fire on them. Watch out especially for the ships to the left and right which, although they start farther away than the ships above and below, move eight times as fast. Good luck, and good hunting.

Program Line No.	Description
1-10	Go to the subroutines for redefining the character set and initializing TextPlot.
20	Initialize graphics, set character base address to redefined character set.
30	Initialize variables.
40-100	Draw the fortress and background.
110-120	Print "SCORE 000" on the screen.
130-170	Erase last gun position.
180-220	Read current joystick position.
230-280	Aiming and firing sequence. The gun is drawn in the new position, and the laser is fired. If the ship is hit, it explodes.
290-310	Updates the score on the screen, digit by digit. Jumps to the end of the game on high score.
320-350	If a ship was destroyed, then uses the random number generator to decide whether a new ship is to be launched. The starting position of the new ship is moved closer to the fortress as the score increases.
360-400	Moves each ship toward the fortress. If the fortress is hit by a ship, then jumps to the end of game routine.
500-620	End of game routine when fortress is destroyed.
700-710	End of game routine on winning game.
20000-20430	Subroutine for TextPlot.
32000-32109	Subroutine for redefining character set.

Variables

SC = Score J = joystick position

J1 = 1,2,3,4 depending on joystick position
MX1 to MX4 = X coordinate of enemy ships
MY1 to MY4 = Y coordinate of enemy ships
M1 to M4 = status of enemy ships; = 0 when ship is blown up;
= 1 when ship is intact
Starx,Stary = X and Y coordinates of stars
ML = memory location
START = byte address of RAMTOP
Z,Y,STAR,N,W,I = loop variables.

Program.

```
1 GOSUB 32000:CLR
10 GOSUB 20000
20 GRAPHICS 7+16:POKE 756,PEEK(106)+1
30 SETCOLOR 2,3,4:SC=0:J1=1:MX1=0:MY1
   =0:MX2=0:MY2=0:MX3=0:MY3=0:MX4=0:M
   Y4=0:M1=0:M2=0:M3=0:M4=0
40 COLOR 1:FOR Y=35 TO 45:PLOT 72,Y:D
   RAWTO 95,Y:NEXT Y
41 COLOR 2
50 PLOT 72,35:DRAWTO 69,32:PLOT 73,35
   :DRAWTO 69,32:PLOT 72,36:DRAWTO 69
   ,32
60 PLOT 72,45:DRAWTO 69,48:PLOT 73,45
   :DRAWTO 69,48:PLOT 72,44:DRAWTO 69
   ,48
70 PLOT 95,35:DRAWTO 98,32:PLOT 94,35
   :DRAWTO 98,32:PLOT 95,36:DRAWTO 98
   ,32
80 PLOT 95,45:DRAWTO 98,48:PLOT 94,45
   :DRAWTO 98,48:PLOT 95,44:DRAWTO 98
   ,48
90 FOR STAR=1 TO 80:STARX=RND(0)*158+
   1:STARY=RND(0)*94+1:PLOT STARX,STA
   RY:NEXT STAR
100 COLOR 0:FOR X=73 TO 94 STEP 2:PLO
    T X,40:NEXT X
110 D=USR(1536,83,3,0,0):D=USR(1536,6
    7,3,1,0):D=USR(1536,79,3,2,0)
120 D=USR(1536,82,3,3,0):D=USR(1536,6
    9,3,4,0):D=USR(1536,48,3,1,8):D=U
    SR(1536,48,3,2,8):D=USR(1536,48,3
    ,3,8)
130 ON J1 GOTO 140,150,160,170
140 D=USR(1536,32,1,70,24):GOTO 180
150 D=USR(1536,32,1,72,34):GOTO 180
160 D=USR(1536,32,1,70,43):GOTO 180
170 D=USR(1536,32,1,68,34)
180 J=STICK(0):IF J=15 THEN GOTO 290
190 IF J=10 OR J=14 OR J=6 THEN J1=1:
    D=USR(1536,16,1,70,24):GOTO 230
```

```
200 IF J=7 THEN J1=2:D=USR(1536,17,1,
    72,34):GOTO 230
210 IF J=5 OR J=13 OR J=9 THEN J1=3:D
    =USR(1536,18,1,70,43):GOTO 230
220 IF J=11 THEN J1=4:D=USR(1536,19,1
    ,68,34)
230 COLOR 1:SOUND 0,25,10,8:SOUND 1,2
    8,10,8:ON J1 GOTO 250,260,270,280
250 PLOT 84,27:DRAWTO 84,0:COLOR 0:PL
    OT 84,27:DRAWTO 84,0:IF M1=1 THEN
     M1=0:D=USR(1536,15,3,MX1,MY1):SC
    =SC+2
255 GOTO 290
260 PLOT 104,40:DRAWTO 159,40:COLOR 0
    :PLOT 104,40:DRAWTO 159,40:IF M2=
    1 THEN M2=0:D=USR(1536,15,3,MX2,M
    Y2):SC=SC+2
265 GOTO 290
270 PLOT 84,54:DRAWTO 84,95:COLOR 0:P
    LOT 84,54:DRAWTO 84,95:IF M3=1 TH
    EN M3=0:D=USR(1536,15,3,MX3,MY3):
    SC=SC+2:GOTO 290
280 PLOT 63,40:DRAWTO 0,40:COLOR 0:PL
    OT 63,40:DRAWTO 0,40:IF M4=1 THEN
     M4=0:D=USR(1536,15,3,MX4,MY4):SC
    =SC+2
290 SOUND 0,0,0,0:SOUND 1,0,0,0:SOUND
     3,0,0,0:IF SC>999 THEN GOTO 700
300 V1=INT(SC/100):V2=INT(SC/10-V1*10
    ):V3=SC-V1*100-V2*10:V1=V1+48:V2=
    V2+48:V3=V3+48
310 D=USR(1536,V1,3,1,8):D=USR(1536,V
    2,3,2,8):D=USR(1536,V3,3,3,8)
320 IF M1=0 THEN IF INT(RND(0)*2+1)=1
     THEN M1=1:MX1=70:MY1=SC/75:D=USR
    (1536,20,2,MX1,MY1)
330 IF M2=0 THEN IF INT(RND(0)*2+1)=1
     THEN M2=1:MX2=79-SC/400:MY2=33:D
    =USR(1536,21,2,MX2,MY2)
340 IF M3=0 THEN IF INT(RND(0)*2+1)=1
     THEN M3=1:MX3=70:MY3=70-SC/75:D=
    USR(1536,22,2,MX3,MY3)
```

```
350 IF M4=0 THEN IF INT(RND(0)*2+1)=1
     THEN M4=1:MX4=60+SC/400:MY4=32:D
    =USR(1536,23,2,MX4,MY4)
360 IF M1=1 THEN D=USR(1536,20,0,MX1,
    MY1):MY1=MY1+1:D=USR(1536,20,2,MX
    1,MY1):IF MY1>=24 THEN GOTO 500
370 IF M2=1 THEN D=USR(1536,21,0,MX2,
    MY2):MX2=MX2-1:D=USR(1536,21,2,MX
    2,MY2):IF MX2<=72 THEN GOTO 500
380 IF M3=1 THEN D=USR(1536,22,0,MX3,
    MY3):MY3=MY3-1:D=USR(1536,22,2,MX
    3,MY3):IF MY3<=43 THEN GOTO 500
390 IF M4=1 THEN D=USR(1536,23,0,MX4,
    MY4):MX4=MX4+1:D=USR(1536,23,2,MX
    4,MY4):IF MX4>=68 THEN GOTO 500
400 GOTO 130
500 SOUND 0,50,8,8:SOUND 1,100,8,8:SO
    UND 2,200,8,8:SOUND 3,5,8,8
510 D=USR(1536,15,3,68,34):D=USR(1536
    ,15,3,70,43):D=USR(1536,15,3,72,3
    4):D=USR(1536,15,3,70,24)
520 D=USR(1536,15,3,69,36):D=USR(1536
    ,15,3,69,40):D=USR(1536,15,3,70,3
    0):D=USR(1536,15,3,71,27)
530 FOR N=0 TO 3:SOUND N,0,0,0:NEXT N
550 FOR N=0 TO 3:SOUND N,N*80+5,8,8:N
    EXT N
560 COLOR 3:PLOT 84,40:DRAWTO 84,20:D
    RAWTO 84,60:PLOT 84,40:DRAWTO 114
    ,40:DRAWTO 54,40:PLOT 84,40:DRAWT
    O 114,20
570 PLOT 84,40:DRAWTO 114,60:PLOT 84,
    40:DRAWTO 54,60:PLOT 84,40:DRAWTO
     54,20
580 FOR W=0 TO 15:FOR W1=1 TO 20:SETC
    OLOR 2,W,5:NEXT W1:NEXT W
585 FOR N=0 TO 3:SOUND N,0,0,0:NEXT N
590 FOR I=1 TO 30:FOR J=1 TO 10*RND(1
    ):SOUND 0,I+10*RND(1),10,8:NEXT J
    :NEXT I:SOUND 0,0,0,0
600 GRAPHICS 2+16:? #6;"GAME OVER..FI
    NAL":? #6;"SCORE ";SC:? #6;"TO PL
```

```
    AY AGAIN":? #6;"PRESS TRIGGER"
610 IF STRIG(0)=1 THEN GOTO 610
620 GOTO 20
700 GRAPHICS 2+16:? #6;"GOOD GAME!!!"
    :? #6;"YOU WON!!! ":? #6;"YOUR S
    PACE FORTRESS":? #6;"SURVIVED"
710 ? #6;"TO PLAY AGAIN":? #6;"PRESS
     TRIGGER ":GOTO 610
19999 END
20000 ML=1536:FOR I=0 TO 252:READ A:P
      OKE ML+I,A:NEXT I:RETURN
20010 DATA 104,240,10,201,4,240
20020 DATA 11,170,104,104,202,208
20030 DATA 251,169,253,76,164,246
20040 DATA 104,133,195,104,201,128
20050 DATA 144,4,41,127,198,195
20060 DATA 170,141,250,6,224,96
20070 DATA 176,15,169,64,224,32
20080 DATA 144,2,169,224,24,109
20090 DATA 250,6,141,250,6,104
20100 DATA 104,141,251,6,104,104
20110 DATA 141,252,6,14,252,6
20120 DATA 104,104,141,253,6,133
20130 DATA 186,166,87,169,10,224
20140 DATA 3,240,8,169,20,224
20150 DATA 5,240,2,169,40,133
20160 DATA 207,133,187,165,88,133
20170 DATA 203,165,89,133,204,32
20180 DATA 228,6,24,173,252,6
20190 DATA 101,203,133,203,144,2
20200 DATA 230,204,24,165,203,101
20210 DATA 212,133,203,165,204,101
20220 DATA 213,133,204,173,250,6
20230 DATA 133,187,169,8,133,186
20240 DATA 32,228,6,165,212,133
20250 DATA 205,173,244,2,101,213
20260 DATA 133,206,160,0,162,8
20270 DATA 169,0,133,208,133,209
20280 DATA 177,205,69,195,72,104
20290 DATA 10,72,144,8,24,173
20300 DATA 251,6,5,208,133,208
20310 DATA 224,1,240,8,6,208
```

```
20320 DATA 38,209,6,208,38,209
20330 DATA 202,208,228,104,152,72
20340 DATA 160,0,165,209,145,203
20350 DATA 200,165,208,145,203,104
20360 DATA 168,24,165,203,101,207
20370 DATA 133,203,144,2,230,204
20380 DATA 200,192,8,208,183,96
20390 DATA 169,0,133,212,162,8
20400 DATA 70,186,144,3,24,101
20410 DATA 187,106,102,212,202,208
20420 DATA 243,133,213,96,0,1
20430 DATA 28
32000 POKE 106,PEEK(106)-16:GRAPHICS
      0:START=(PEEK(106)+1)*256:POKE
      756,START/256:POKE 752,1
32020 POKE 559,0:FOR Z=0 TO 1023:POKE
       START+Z,PEEK(57344+Z):NEXT Z:R
      ESTORE 32100
32025 FOR I=1 TO 30:FOR J=1 TO 10*RND
      (1):SOUND 0,I+10*RND(1),10,8:NE
      XT J:NEXT I:SOUND 0,0,0,0
32030 READ X:IF X=-1 THEN RESTORE :PO
      KE 559,34:RETURN
32040 FOR Y=0 TO 7:READ Z:POKE X+Y+ST
      ART,Z:NEXT Y:GOTO 32030
32100 DATA 632,145,82,44,222,57,52,74
      ,137
32101 DATA 640,24,24,24,60,126,126,60
      ,255
32102 DATA 648,128,176,248,255,255,24
      8,176,128
32103 DATA 656,255,60,126,126,60,24,2
      4,24
32104 DATA 664,1,13,31,255,255,31,13,1
32105 DATA 672,231,231,126,60,24,24,2
      4,0
32106 DATA 680,3,7,15,252,252,15,7,3
32107 DATA 688,24,24,24,24,60,126,231
      ,231
32108 DATA 696,192,224,240,63,63,240,
      224,192
32109 DATA -1
```

High Speed Animation With Character Graphics

Charles Brannon

Sound, color, and detail are all important in an arcade-style game, but movement generates the most interest. It's programmed into the brain. In all vision oriented creatures, motion takes precedence over all other visual stimulation. A frog notices a fly not because of any characteristic shape, or even its buzz, but because of its movement.

In a good arcade game, there is a lot of motion. Alien ships swarm and dive, invaders weave back and forth, ghosts chase, balls bounce – the mere description sounds exciting. If we want to develop a thrilling game, we must be able to quickly move objects around on the screen.

Speed is the key word. Let's face it – arcade games require fast animation of many objects. True arcade-quality games almost always require a high-speed language such as FORTH or machine language.

But BASIC is *not* a high-speed language. Its generality, style, and ambiguity make it easy to learn and use, but hard for a computer to efficiently execute. However, since BASIC is easier to use and modify for most people, let's see what we can do to make the most of what we've got.

Optimizing BASIC

Remember that computers are rather simple-minded, so try to break your task into small pieces. Try to have BASIC do as little work as possible. For example, to create a glowing image, we just need to change its colors rapidly. For example, to flash something drawn in in COLOR 1, we could code:

```
100 FOR I=1 TO 100
110 SETCOLOR 0,INT(16*RND(0)),INT(16*
```

```
     RND(0))
120 NEXT I
```

The FOR/NEXT loop controls the duration of the flashing effect, but line 110 is the heart. It picks a random color and a random luminance for COLOR 1 (color register zero). To speed this up, we could POKE directly into the color register. There is one memory location (for our purposes) for each color register. These are located from 704-707 (the player/missile color registers) and from 708 to 712 (COLOR 0 to COLOR 4). Each can hold an even number from 0-254, according to the formula:

COLORBYTE = HUE*16 + LUMINANCE

So we could speed up line 110 by using:

```
110 POKE 708,INT(16*RND(0))*16+INT(16
    *RND(0))
```

This is still rather slow, since BASIC must interpret the long expression of INT's and RND's and perform three multiplications. Fortunately, there is another memory location called RANDOM ($D20A, or 53770 in decimal) that will give us a random number from 0-255. We can read RANDOM with PEEK(53770), and POKE it directly into color register zero:

```
110 POKE 708,PEEK(53770)
```

We've reduced a slow, albeit more readable and understandable, BASIC instruction with a direct POKE to a color register. POKEs are the key to fast graphics, then.

Animation

What about animation? The first thing that comes to mind is the technique of drawing a figure on a high-resolution screen such as GRAPHICS 7, redrawing it at a new location, and then erasing the old image. The following routine will move a box from left to right in GRAPHICS 7:

```
100 GRAPHICS 7
110 FOR I=1 TO 100
120 COLOR 1:GOSUB 200:REM DRAW A BOX
130 COLOR 0:GOSUB 200:REM ERASE IT
140 NEXT I
150 END
199 REM DRAWS A BOX:
```

```
200 PLOT I,0:DRAWTO I+10,0
210 DRAWTO I+10,10:DRAWTO I,10
220 DRAWTO I,0:RETURN
```

As you can see, the motion is smooth, but slow and flickery. This is unavoidable in BASIC. BASIC just can't draw things fast enough to provide fast animation.

What About P/M Graphics?

The Atari solution to high-speed animation is player/missile graphics. These shapes can be moved over the playfield without erasing what they pass over, unlike the method mentioned above. Unfortunately, P/M graphics are not suitable for purely BASIC programs, unless you want only horizontal movement. Vertical motion in BASIC is also slow and flickery.

One solution is machine language, and you will find some machine language routines for using P/M graphics from BASIC in this book. But if you're not ready for machine language, can't quite grasp P/M graphics, or if you're looking for an easier way to quickly move dozens of objects, read on.

POKEing Graphics

What if we could use POKE to generate graphics? That way, we could simplify graphics the same way we did the color registers. Did you ever wonder how the Atari displays a screen? It's a complex subject, but it can be simplified:

COMPUTER→MEMORY→ANTIC/CTIA→TV

When the computer wants to display something, say the character "A", it places a number representing "A" into memory. The ANTIC chip, a video microprocessor, continually looks at the memory, calculates a TV display, and sends this information to the CTIA (or GTIA in recent Ataris), which draws the picture on the television screen.

The computer could directly tell the CTIA what to do, but if it did bypass ANTIC, the computer would be responsible for all display, leaving little or no time for its main job – computing! So the memory is like an image of the TV screen. Since the computer can both read and write to memory, it can place characters on the screen by writing to this memory, and can look at the TV indirectly by reading this memory.

Perhaps our scheme is becoming clear. Instead of using PLOT, DRAWTO, or POSITION and PRINT (which are slow,

slow, slow), we can use POKE to directly place a character on the screen. To move the letter "A" across the screen, we just POKE a 33 (the value corresponding to an "A") into the screen memory, and erase it by replacing it with a space. A fragment of code might look like this:

```
SCREEN=PEEK(88)+256*PEEK(89)
FOR I=1 TO 39:POKE SCREEN+I,33:POKE S
CREEN+I-1,0:NEXT I
```

This would move the letter "A" left to right, assuming the variable SCREEN has been set to the start of screen memory. Notice how fast the character moves. Imagine that the "A" is a spaceship, or an alien invader. That's some fast animation!

Defeat The Invading Q's!

But an "A" is *not* an alien invader, and it isn't much fun shooting at letters of the alphabet. Fortunately, there are two solutions. First, we can use some special characters built into the Atari. For example, press CTRL-T (hold down the CTRL key and press "T"). You'll see a "ball" character. This could be used as a ball in a pong-type game.

Next, try CTRL-A, CTRL-T, and CTRL-D, in that order. You should see a "tie fighter," right out of *Star Wars*. You can see the potential here, but it's not easy to get much color from a GRAPHICS 0 display. Instead, you can use these *character graphics* in graphics modes one and two, which generate large-size, five-color text. It's rather complicated to use the CTRL graphics in these modes, but another option is available.

Custom Characters

As you are probably aware, you can rather easily redefine any character. You could, for example, change the alphabet into a foreign-language alphabet, such as Greek. You could also redefine a character into a spaceship or a blue meanie. These redefined characters can be shaped to resemble almost anything, and then moved about at high speed. The details of customizing the Atari character set are explained in the previous chapter.

Since each character is 8x8 blocks, and takes up only one byte of memory on the screen, you can move up to 64 pixels (8x8) with a single POKE. Using PEEK, you can scan ahead in the direction of movement for a collision with other screen objects, much faster than you could with POSITION and LOCATE statements.

Finding The Screen

For the sake of example, let's use a GRAPHICS 1 screen. The computer must store the information for the screen somewhere in its memory. Since this is the same memory that you use to store programs or other information, the screen memory must be somewhere out of the way. The GRAPHICS command always places the screen where it perceives the top of memory to be. If you have a 32K machine, the top of memory is at location 32767 (32*1024-1). On an 8K machine, the screen would reside at just under the 8191 byte limit. Furthermore, the screen memory starts at different places in each graphics mode. For example, GRAPHICS 8 must start the screen 8,000 bytes from the top of memory.

Locating the starting address of screen memory would seem to be rather tricky. Sure, you could probably look the value up in a large table (indexed by memory size and graphics mode), but why not just ask the computer?

A Double Byter

As it turns out, the starting address of screen memory is stored at locations 88 and 89. Since a memory location can hold only a number from 0-255, it is obvious why two memory locations are needed to hold numbers as large as 65,535. The two locations each hold a part of the number. So, we can use PEEK to calculate the starting address of the screen:

```
SCREEN=PEEK(88)+256*PEEK(89)
```

The second part of the number is always multiplied by 256 and added to the first. There are many of these double-byte locations in the Atari, and you use the same formula to read them.

Now that we have the screen address, we're ready to begin. Note that you should place the screen calculation line *after* the GRAPHICS statement in your program.

A Better Way

Formerly, you probably used something like POSITION X,Y:PRINT#6;"A" or COLOR 65:PLOT X,Y to place a character on a graphics mode 1 or 2 display, at horizontal location X, and vertical location Y. For GRAPHICS 1, X can range from 0 to 19, and Y ranges from 0-23. It's almost that simple to POKE a value into screen memory, using:

```
POKE SCREEN+X+20*Y,33
```

(SCREEN is the address of screen memory)

The vertical component, Y, is multiplied by 20. Think of screen memory as 24 rows of 20 characters. You could go strictly left to right, from 0 to 19, then wrap around to 20 through 39, 40 through 59, etc. But if you want to access the screen by X,Y coordinates, you multiply Y by 20 to reach line Y. You can see that the last memory location would be at X,Y:19,23. Using "X+20*Y", this would give us 479. So to place an "A" at the top left corner of the screen (*home*, or the origin), use POKE SCREEN,33. To place a "B" at the lower right-hand corner, then, we could use POKE SCREEN+19+20*23,34 or POKE SCREEN +479,34.

Internal Vs. ASCII

If you already happen to know the ASCII value of "A", which is 65, you may wonder why we POKEd the screen with a 33. The reason is that although you use ASCII to PRINT characters to the screen, printer, or disk drive, the Atari uses an internal character set for its own uses. Two of these uses are for character set look-up and for storing information in screen memory.

Why doesn't the Atari use ASCII internally? Well, graphics modes one and two permit you to display four colors of characters. This requires two bits to hold the colors from 0-3. These bits are stored as part of the character's numeric, or binary, value. This leaves only six bits for the character, restricting it to the range of 0-63. Whew! This is only half of a full 128-byte character set, so Atari restructured the order of the characters so that uppercase and punctuation are in the top half, and lowercase and graphics in the lower, a feat not possible with standard ASCII. This way, with a "switch" POKE, you could use either half in graphics mode one or two. (The lower half is used in a demo in the *Atari BASIC Reference Manual*, "Seagull over Ocean," on page H-11, and in our example program at the end of this article.)

The result of this is that we must translate between ASCII and the internal character set. You can compare the two using Figure 1, or look up the number of a character in Table 1. As "A" is 65 in ASCII and 33 internally, you may be tempted to think the translation is as simple as subtracting 32 from the ASCII value. This works fairly well in graphics modes 1 and 2; but, as you can see from Figure 1, lowercase doesn't move at all, and you must add 64 to convert the "control graphics"

characters from ASCII (also called ATASCII for ATari ASCII) to the internal character set.

Action!

Let's look at a fragment of code that moves the character "A" diagonally from top left towards bottom right. Again, the "A" could be redefined with a custom character set, or you could use another character by looking it up in Table 1 and replacing the 33 below:

```
10 GRAPHICS 1+16:REM Full screen
20 SCREEN=PEEK(88)+256*PEEK(89)
30 FOR I=0 TO 19
40 POKE SCREEN+I+20*I,33
50 NEXT I
60 GOTO 60:REM Wait for [BREAK]
```

The sample program should be fairly self-explanatory. The "I" index ranges from 0 to 19, which is used to select both a horizontal and vertical value. The "33" places an "A" on the screen. This is not erased, so we get a diagonal line of "A's". Add this line to erase an "A" after it's drawn. This creates the illusion of movement.

```
45 POKE SCREEN+I+20*I,0
```

A Star Maker

Let's add a line that litters the screen with stars, as in an outer space scene. The code for our star will be 14, the period:

```
22 FOR I=1 TO 50
24 POKE SCREEN+INT(480*RND(0)),14
26 NEXT I
```

We don't care to independently control both the X and Y coordinates of each star. We just want to pick a random screen location from 0-479 (0,0 to 19,23). The above piece of code will place about 50 stars on the screen. There could be fewer than 50 stars, even though the index ranges from 1 to 50. Can you see why? Imagine what would happen if the same random number were picked twice within the loop. The second star would simply replace the first. We could make our star-maker fragment more intelligent by having it look to see if there is already a star where it wants to place one.

Taking A PEEK

With PLOT, you can use LOCATE, to "read" a point on the screen. This is much slower than PEEK. We use the same formula as POKE to read point X,Y (since we're looking at the same memory): Z=PEEK(SCREEN+X+20*Y). Of course, the variables X,Y, and Z are entirely arbitrary. We can change line 24 to look at the screen before it plunks down a star, and force it to find a blank spot:

```
24 R=SCREEN+INT(480*RND(0)):IF PEEK(R
   )<>0 THEN 24
25 POKE R,14
```

We could likewise place a lookahead statement into the loop that moves the "A" to see if it hits a star:

```
35 IF PEEK(SCREEN+I+20*I)<>0 THEN 60

60 POKE SCREEN+I+20*I,10
70 GOTO 70
```

We place this statement at line 35, before line 40, for if we placed it after line 40, the PEEK would read the "A" we just POKEd in line 40, and would of course detect a collision. As is, it checks the intended position first, and if there is something other than a space (code 0), there must be a star there, so we exit to line 60, which places an asterisk, representing an explosion, at the intended position.

If we didn't want to explode when we collided with a star, we could have left out line 35. When the "A" traveled over a star, it would merely erase it. If we wanted to have the "A" just pass over the star nondestructively, we could use PEEK to read the contents of the next position of the "A", save it in a variable, and then restore the old value when we're ready to move the "A" to its next position:

```
35 P=PEEK(SCREEN+I+20*I)
45 POKE SCREEN+I+20*I,P   (instead of 0)
60 Delete this line if you changed it above.
```

A Sample Program

A complete program using all these techniques is presented at the end of this article (Program 1). Using GRAPHICS 1, it draws a border around the screen. If every animated object checks for

a collision with the border, we won't have to worry about errant creatures flying off the screen (out of the boundaries of screen memory) into our program's memory. If you don't use a border in your game, you should make sure that any POKE is within the range of SCREEN + 0 to SCREEN + 479 for GRAPHICS 1. (You can refer to Table 2 for screen limits for other graphics modes.)

Two objects are moved, the player (moved with the joystick), and a bouncing ball. The game is written using the lower half of the character set. SETCOLOR 0,0,0 is used to make the hearts (which map into the same area as SPACE does in the top half) disappear. POKE 756,226 selects the bottom half.

The ball is CTRL-T, but the player can be any of eight characters, depending on which direction he is facing. Only one character representing the character is used at a time, however.

Four-Color Mapping

Figure 1 (Internal Character Set) comes in handy for figuring out what colors a character will be. For example, you know that you can get four colors of "A" by using "A", lowercase "a", inverse video "A", and inverse video lowercase "a". Not so obvious is, say, the dollar sign. There's no such thing as a lowercase dollar sign, but you can still get four colors.

Just count down two rows to see the character you should use. If you count down two rows from "A", you will find a lowercase "a". Two rows down from the dollar sign is CTRL-D. Incidentally, we can tell that the strange graphics symbol two rows down from the dollar sign is indeed a CTRL-D, since it is underneath the letter "D".

If we are using the second half of the character set for our game, we can use the same figure (Figure 1) to see where the strange graphics characters map out. To get a blue club, we just use an inverse-video "0". (See how the "0" is two rows above the club, CTRL-P?) I'm sure you will find many uses for Figure 1 in the future.

Analyzing The Sample Game

Let's look at Program 1 together. In addition to using character graphics for animation, this program uses a few programming tricks that require explanation.

Initialization

Lines 100-180 initialize the direction and character arrays (de-

scribed later). Lines 190-230 set up the GRAPHICS 1 screen, select the lower half of the character set, and set color register zero to black in order to erase the inevitable hearts. Line 240 defines variables for two characters used, the ball and the star (which is CTRL-I, a small block).

Lines 250-370 draw a border around the screen (Lines 270-300 draw the sides, 310-350 the top and bottom, and 370 the corners). Lines 390-400 scatter stars upon the screen. The IF statement checks to see if the intended spot is empty before POKEing a star. This prevents a star from overwriting the screen border just drawn.

The variables PX and PY, initialized in line 410, hold the player's horizontal and vertical position, respectively (1-18 and 1-22). Line 410 also places the upward-facing player character on the screen. Line 420 initializes the ball X,Y variables and effectively selects a random starting location for the ball.

Line 410 sets the ball's horizontal and vertical displacement variables. Each move, DX is added to BX, and DY to BY. If DX and DY are 1, then the ball will move diagonally towards the lower right (+1 right, +1 left). DX and DY can take on other values from -1,0, and +1. In this program, either DX, DY, or both DX and DY are switched (their sign is changed, as in DX=-DX) to bounce the ball in an opposite direction when it hits the wall or a star.

The Main Loop

The main program, in the form of a continuous loop, goes from 460-770. Lines 460-600 let the player move, if he wants to (the joystick is pushed); otherwise the ball moves.

Temporary variables hold the updated values of PX and PY. These variables are changed by the arrays DX and DY, which, like DX and DY for the ball, make the player move up, down, left, or right. ST, the joystick position, is used as an index into the array, instantly selecting the proper displacement. For example, a joystick reading of 14 means *up*. DY(14) equals -1, and DX(14) is 0. This subtracts one from the player Y value (moving it up), and leaves the horizontal value unchanged. If DX was non-zero, a diagonal motion would result. Puzzle over the concept of displacement, but remember the technique. Using an array as a look-up table saves you from having to use a list of IF/THEN statements such as IF ST=14 THEN PY=PY-1:GOTO xxx. This saves memory, and, more importantly,

time. Table lookup gives BASIC less work to do and results in some very fast games.

Collision Detection

Line 510 calculates PPOS, which is the absolute memory address of the player in screen memory. We can then easily check for a collision, as in lines 520-530. Remember, the player hasn't moved yet, since we've only used temporary variables. We're just checking the anticipated move. Since the ball and the star have already been processed, if there is any other character detected in line 540, it must be one of the characters in the boundary. If so, we cannot let the player advance, since this would erase the boundary and let him escape from the screen. (Shades of *TRON*!) Since PX and PY haven't been changed yet, we just exit and let the ball move. The player hasn't gone anywhere.

Assuming the player makes a legal move, table lookup is used once again to find the character corresponding to the direction the player is facing. The characters used are arrows and the diagonal corner characters. The previous player character, still pointed to by PX and PY, is erased in line 580, and the new character is POKEd into PPOS, the updated location. Now PX and PY are changed to reflect the new location (line 600).

Moving The Ball

Lines 640-730 move the ball. This is a fairly simple routine. All we do is update the X and Y variables (using temporary variables as we did with the player), with DX and DY. The ball's position in screen memory is calculated in line 650. Line 660 checks for a collision with the player character, meaning that the ball is caught. Since there are eight player characters, we could need up to eight IF clauses. But since the only thing drawn in COLOR 2 (green by default) is the character, all we need do is check for a character in the range of 64-96. (All characters from 64-96 are drawn in COLOR 2.)

If the ball hits anything else, lines 700-720 reverse the ball's direction. The RND (RaNDom) statements insure that the ball won't get caught in an endless ricochet loop. Finally, lines 740 to 760 update the ball, as we did with the player. It's a good thing that the computer executes all this faster than we can read about it!

Game Over

The two exit routines at lines 1000 and 2000 either cheer the player for catching the ball or result in his explosion if he collides with a star. Line 1040 flips the ball character with the cross (CTRL-S) character. Line 1050 is the heart of the sound effect. Line 1070 wraps up the sound effects for the win routine. There is no scoring in this game; you either win or lose. You can add features to the game, of course. But remember, since we're using the lower half of the character set, there aren't any numerals to print a score with.

Line 2000 explodes the character and produces the explosion sound. The high-speed color POKE is used extensively to provide flashing (lines 2040 and 2090). The explosion effect is produced by POKEing the graphics characters from CTRL-A to CTRL-F into the player's position. This doesn't look too much like an explosion, but it does produce a flickering effect. If you expanded this program into a full-fledged game, you could add an option for three lives here.

Custom Characters Are More Fun

I previously mentioned that a custom character set adds flair to a program like this. To avoid confusion, I refrained from using a custom character set in Program 1. You should type in and try to understand Program 1 before adjusting it with Program 2, which adds custom characters to the game. The changes to Program 1 given in Program 2 are trivial, but they make the game much more fun. Good, detailed graphics really jazz up a game.

The subroutine at line 5000 places the character set in memory, just below the screen. The character data is in the DATA statements from 5005-5024. It takes several seconds to initialize the character set. However, since RUN does not clear out this memory, we can check to see if the character set has already been initialized, as in line 5001. Subsequent RUNs start instantly.

The up/down/left/right arrows have been redefined as a sort of cup, or scoop, that you use to catch the ball. Since diagonal movement is permitted in the game, the corner characters (CTRL Q,E,Z and C) are redefined as diagonal scoops. Unfortunately, these same characters are also used as the corners of the screen border. To avoid too much modification of Program 1, the scoops are used as the corners of the border

Table 1. Internal Character Set.

0		16	0	32	@	48	P	64	♥	80	♣	96	♦	112	p
1	!	17	1	33	A	49	Q	65	├	81	┌	97	a	113	q
2	"	18	2	34	B	50	R	66	│	82	─	98	b	114	r
3	#	19	3	35	C	51	S	67	┘	83	┼	99	c	115	s
4	$	20	4	36	D	52	T	68	┤	84	●	100	d	116	t
5	%	21	5	37	E	53	U	69	┐	85	▬	101	e	117	u
6	&	22	6	38	F	54	V	70	/	86	|	102	f	118	v
7	'	23	7	39	G	55	W	71	\	87	┬	103	g	119	w
8	(	24	8	40	H	56	X	72	◢	88	┴	104	h	120	x
9	)	25	9	41	I	57	Y	73	▗	89	▌	105	i	121	y
10	*	26	:	42	J	58	Z	74	◣	90	└	106	j	122	z
11	+	27	;	43	K	59	[	75	▝	91	⅋	107	k	123	♠
12	,	28	<	44	L	60	\	76	▘	92	↑	108	l	124	|
13	-	29	=	45	M	61	]	77	▔	93	↓	109	m	125	π
14	.	30	>	46	N	62	^	78	▁	94	←	110	n	126	◀
15	/	31	?	47	O	63	_	79	▖	95	→	111	o	127	▶

anyway, and the connecting characters (CTRL-R and SHIFT =) are thickened.

Finally, five characters have been defined for the explosion of the player. These depict a small center core expanding outward in five stages. When POKEd rapidly into the same spot, they create a good explosion effect. Since CTRL-A through CTRL-F (used in Program 1 for the explosion) already contain custom characters, five characters from CTRL-J to CTRL-N were used instead, and line 2030 changed appropriately.

The speed of PEEKs and POKEs lets you program some really fast games in BASIC. The more objects you move, the slower the game will run, of course. But you may very well find that you must insert delay loops just to slow down the game enough to play it!

Figure 1. Atari Character Sets.

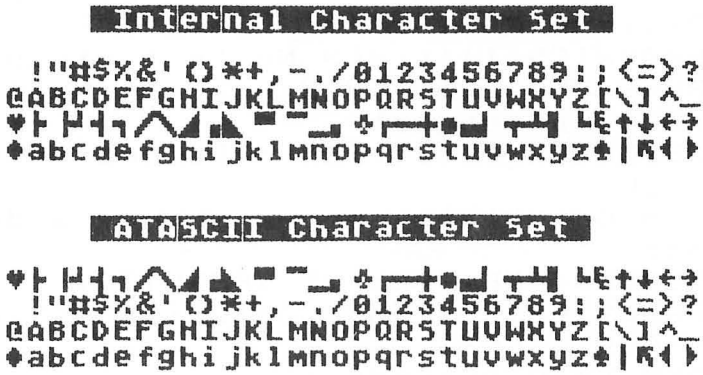


Table 2. Screen Limits and Characteristics.

Mode	X-Limit	Y-Limit	Range	Bytes/Line
0	39	23	0-959	40*
1	19	23	0-479	20
2	19	11	0-239	20

Assumes full screen mode (no text window).

*This is the number in the formula:
SCREEN+X+20*Y. Use SCREEN+X+40*Y for GRAPHICS 0.

Program 1. Sample Program Using PEEK and POKE for Fast Graphics.

```
100 REM Program demonstrates
110 REM Animation by PEEKing
120 REM and POKEing.
130 DIM CHR(15),DX(15),DY(15)
140 REM Direction offsets for each jo
    ystick position
150 DX(14)=0:DX(13)=0:DX(9)=-1:DX(10)
    =-1:DX(11)=-1:DX(5)=1:DX(6)=1:DX(
    7)=1
160 DY(11)=0:DY(7)=0:DY(6)=-1:DY(10)=
    -1:DY(14)=-1:DY(5)=1:DY(9)=1:DY(1
    3)=1
170 REM The character for each joysti
    ck position
180 CHR(14)=92:CHR(13)=93:CHR(11)=94:
    CHR(7)=95:CHR(10)=81:CHR(6)=69:CH
    R(9)=90:CHR(5)=67
190 GRAPHICS 1+16:REM No text window
200 POKE 756,226:REM Use lower half o
    f character set
210 SETCOLOR 0,0,0:REM Make hearts va
    nish
220 REM Calculate screen memory addre
    ss:
230 SCREEN=PEEK(88)+256*PEEK(89)
240 BALL=20+128:STAR=73+128:REM CTRL-
    T AND CTRL-I
250 REM Draw a border:
260 BARHORIZ=82+128:REM Horizontal ba
    r (CTRL-R)
270 FOR I=1 TO 18
280 POKE SCREEN+I,BARHORIZ
290 POKE SCREEN+460+I,BARHORIZ
300 NEXT I
310 BARVERT=124+128:REM Vertical bar
    (SHIFT-EQUALS)
320 FOR I=1 TO 22
330 POKE SCREEN+I*20,BARVERT
340 POKE SCREEN+19+I*20,BARVERT
350 NEXT I
```

```
360 REM Do corners CTRL-Q,CTRL-E,CTRL
    -Z,CTRL-C
370 POKE SCREEN,81+128:POKE SCREEN+19
    ,69+128:POKE SCREEN+460,90+128:PO
    KE SCREEN+479,67+128
380 REM Put in random "stars"
390 FOR I=1 TO 20:R=SCREEN+INT(480*RN
    D(0)):IF PEEK(R)=0 THEN POKE R,ST
    AR:REM Don't overwrite border
400 NEXT I
410 PX=10:PY=11:POKE SCREEN+PX+20*PY,
    CHR(14):REM Player X, Player Y.
    Put "up-arrow" character on scree
    n
420 BX=INT(18*RND(1)+1):BY=INT(22*RND
    (1)+1):REM Ball X, Ball Y Selecte
    d randomly
430 DX=1:DY=1:REM Direction offsets f
    or ball
440 REM Main loop
450 REM Check for player's move:
460 ST=STICK(0)
470 IF ST=15 THEN 640:REM Let ball bo
    unce
480 REM Temporary variables hold upda
    ted PX and PY
490 TX=PX+DX(ST):REM X offset 1,0,-1
500 TY=PY+DY(ST):REM Same with Y.  Ta
    ble lookup is fast!
510 PPOS=SCREEN+TX+20*TY:REM Calculat
    e current position
520 IF PEEK(PPOS)=BALL THEN 1000:REM
    Hit ball
530 IF PEEK(PPOS)=STAR THEN 2000:REM
    Hit star
540 IF PEEK(PPOS)<>0 THEN 640:REM If
    wall hit, don't let player advanc
    e
550 REM Update player
560 REM Table lookup also replaces IF
    /THEN, and is ultimately more mem
    ory efficient:
```

```
570 CHR=CHR(ST)
580 POKE SCREEN+PX+20*PY,0:REM Erase
    previous character
590 POKE PPOS,CHR:REM Place new one
600 PX=TX:PY=TY:REM Update variables
610 REM
620 REM Let ball have its turn
630 REM
640 TEMPBX=BX+DX:TEMPBY=BY+DY:REM DX
    and DY are direction offsets
650 BPOS=SCREEN+TEMPBX+20*TEMPBY:REM
    Ball absolute position
660 IF PEEK(BPOS)>64 AND PEEK(BPOS)<9
    6 THEN 1000:REM Ball hit player
670 IF PEEK(BPOS)=0 THEN 730
680 REM Any other collision is a boun
    ce
690 REM Change direction of either X,
     Y, or both X and Y:
700 IF RND(1)>0.5 THEN DX=-DX:GOTO 46
    0
710 IF RND(1)>0.5 THEN DY=-DY:GOTO 46
    0
720 DX=-DX:DY=-DY:GOTO 460
730 REM Update ball
740 POKE SCREEN+BX+20*BY,0:REM Erase
    old ball
750 POKE BPOS,BALL:REM Draw new ball
760 BX=TEMPBX:BY=TEMPBY:REM Update ba
    ll variables
770 GOTO 460
1000 REM Hit ball, win!
1010 BALLPOS=SCREEN+BX+20*BY:FLIP=128
     :REM BALL POSITION, "FLIPPING" V
     ARIABLE
1020 REM FLIP is used to alternate tw
     o characters in one spot
1030 REM 257-FLIP switches FLIP from
     128 to 129, and vice-versa
1040 FOR I=1 TO 5:FOR J=1 TO 6:POKE B
     ALLPOS,19+FLIP:POKE 710,PEEK(537
     70)
```

```
1050 FOR W=5 TO 0 STEP -5:SOUND 0,J*2
     ,10,W:NEXT W:FLIP=257-FLIP:NEXT
     J:NEXT I
1060 REM STRANGE SOUND, FLASH BALL AG
     AIN
1070 FOR I=255 TO 0 STEP -5:SOUND 0,I
     ,12,8:SOUND 1,I,10,8:POKE 710,I:
     NEXT I
1080 RUN
2000 REM UH-OH! Hit a star!
2010 FOR I=100 TO 200 STEP 10
2020 XPLODE=XPLODE+0.5:REM Advance ch
     aracter every other "I", from 1-
     5
2030 POKE SCREEN+PX+20*PY,65+INT(XPLO
     DE):REM Place a graphics charact
     er
2040 POKE 709,PEEK(53770):REM More "c
     heap thrill" color
2050 SOUND 0,I,0,15-INT((I-100)/6.66)
     :REM Explosion sound
2060 NEXT I
2070 POKE SCREEN+PX+20*PY,0:REM Erase
      character
2080 REM Flash screen
2090 FOR I=1 TO 100:POKE 711,PEEK(537
     70):NEXT I
2100 RUN
```

Program 2. Make these changes to Program 1 to add custom characters.

```
200 GOSUB 5000:POKE 756,CHSET/256:REM
     activate character set
210 REM DELETE THIS LINE
240 BALL=20+128:STAR=9:REM CTRL-T AND
     CTRL-I
2030 POKE SCREEN+PX+20*PY,74+INT(XPLO
     DE):REM Place an explosion chara
     cter
5000 CHSET=(PEEK(106)-8)*256:FOR I=0
```

```
     TO 7:POKE CHSET+I,0:NEXT I:REM C
     LEAR OUT HEART
5001 RESTORE 5005:IF PEEK(CHSET+9*8)=
     16 THEN RETURN :REM IF CHSET STI
     LL IN MEMORY, WHY RE-INITIALIZE?
5002 READ A:A=A-64:IF A<0 THEN RETURN
5003 FOR J=0 TO 7:READ B:POKE CHSET+A
     *8+J,B:NEXT J
5004 GOTO 5002
5005 DATA 67,248,254,241,225,96,48,24
     ,0
5006 DATA 69,32,64,64,196,228,252,248
     ,240
5007 DATA 73,16,84,40,198,40,84,16,0
5008 DATA 74,0,0,16,56,56,16,0,0
5009 DATA 75,0,32,0,16,8,16,4,0
5010 DATA 76,0,64,0,16,68,16,4,0
5011 DATA 77,0,64,16,2,0,128,32,2
5012 DATA 78,128,8,1,0,0,0,0,64
5013 DATA 79,8,0,0,0,0,0,0,0
5014 DATA 81,4,2,2,35,39,63,31,15
5015 DATA 82,0,255,255,255,255,255,0,
     0
5016 DATA 83,28,34,73,93,93,73,34,28
5017 DATA 84,0,0,8,28,28,8,0,0
5018 DATA 90,31,127,143,135,6,12,24,0
5019 DATA 92,66,195,129,195,231,126,6
     0,24
5020 DATA 93,24,60,126,231,195,129,19
     5,66
5021 DATA 94,60,126,15,7,7,15,126,60
5022 DATA 95,60,126,240,224,224,240,1
     26,60
5023 DATA 124,60,60,60,60,60,60,60,60
5024 DATA 127,16,24,28,30,30,28,24,16
5025 DATA -1
```

Chapter 5

Animation With Player/Missile Graphics

Introduction To Player/Missile Graphics

Bill Wilkinson

This article describes the features of the Atari player/missile graphics system ("P/M graphics" for those of us with lazy typing fingers). Although there are now other systems available with similar capabilities (notably "sprites" on the Commodore and Texas Instruments TI 99-4A computers), there are several aspects of P/M graphics which are uniquely and powerfully Atari.

For reasons having to do with lack of information and (often) an abundance of misinformation, many Atari owners think of players and missiles as some mysterious aspect of the machine which requires convoluted machine language and arcane rites to control properly. In truth, P/M graphics is in many ways less mysterious than the standard Atari "playfield" graphics. Have you yet truly deciphered the relationship between SETCOLOR and COLOR? (I haven't. I usually use trial and error to find the connection that I need.) Have you mastered the concept of display lists? (I didn't ask if you could produce one, just if you understood the level of indirection that is needed to produce even the simplest display on an Atari.) Player/missile graphics is actually *simple* compared to some of these obscurities.

I think the first thing needed to understand P/M graphics is a little flexibility in conceiving how memory is mapped into display in the Atari computer. Consider Figure 1. As far as the Central Processing Unit (CPU) or most of the Atari hardware is concerned, memory is simply one long string of bytes. (Well, some parts of the system like to digest memory in 1K, 2K, or 4K byte blocks, but within those blocks it's all a string of bytes.) But if you look at Figure 1, you will probably soon decide that this is not a reasonable way for human beings to consider mem-

ory, especially human beings who are trying to visualize memory being displayed as graphics.

So consider instead what we know of BASIC graphics mode 19 (GRAPHICS 3 + 16). GRAPHICS 19 (which is simply a full-screen version of GRAPHICS 3) consists of 24 lines of 40 pixels each, where each pixel occupies only two bits of memory, implying that a line is only 10 bytes long. Instead of thinking of memory as one long string of bytes, why not consider it as an array of ten-byte strings? This visualization is presented in Figure 2, and you will probably agree that this is a much clearer representation than that of Figure 1.

One more exercise before leaving this subject: try visualizing the normal text (GRAPHICS 0) display screen as a representation of memory. How many lines are there? How many bytes per line? I hope you answered 24 lines of 40 bytes each; a pictorial representation of this display mode is shown in Figure 3.

So just exactly what is a player or a missile? First, let's note that for most purposes there's no real difference between a player and a missile other than size, so all further references to "players" may be assumed to refer to missiles also, unless otherwise noted. A player, then, is simply the graphic video display of a portion of the Atari computer's main memory. "So what?" you say. "That's how all computers put stuff on the screen: by displaying the memory." True. And we just showed diagrams of how the Atari also displays its main video screen from what Atari calls "playfield memory." But players and missiles are displayed independently of the playfield and from an entirely separate segment of memory.

The "S:" ("Screen") device driver, which is what actually processes such BASIC keywords as GRAPHICS, PLOT, and DRAWTO, knows nothing of P/M graphics. In a sense this is proper: the hardware mechanisms that implement P/M graphics are, for virtually all purposes, completely separate and distinct from the "playfield" graphics supported by "S:". For example, the size, position, and color of players on the video screen are completely independent of the GRAPHICS mode currently selected and any COLOR or SETCOLOR commands currently active. In Atari parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical stripe on the screen.

We again take refuge in a diagram, that of Figure 4. This figure shows a standard playfield display along with that por-

tion of Random Access Memory (RAM) being used to generate the display (and note the representation of RAM as a series of character strings). But notice that the figure also shows another piece of memory being used to display something else on part of the screen. This "something else" is a player. And notice that the player's portion of RAM is shown as an "array" of character strings where each string is only one byte long. This is *always* true: *all* players are always displayed as a one-byte wide array. There are no display lists to worry about, no graphics modes (using 10 or 20 or 40 bytes per line, and which is which?), and no visualization problems. It's like being back to thinking of memory as one long string of bytes – almost.

The "almost" is the kicker. First, note that we need to make sure we are thinking of the string as being stacked vertically. Second, the pictorial representation (the player on the screen instead of the string of bytes in memory) is actually the more accurate one, since each player is always a "semi-fixed" length: Player 1 starts on the very next byte after Player 0, and so on. Third, there are actually two choices open to the user regarding the amount of memory used by each player (hence the words "semi-fixed"): players may have very fine vertical resolution (equivalent to GRAPHICS 8), in which case they occupy 256 bytes each; or they may have relatively coarse resolution (equivalent to GRAPHICS 7), in which case they occupy 128 bytes each. But even with this minor complication, players are fairly simple, since all players must always have the same resolution.

Sounds dull? Consider: each player (and there are four or five, depending on how you think of missiles) may be "painted" in any of the 128 colors available on the Atari (see SETCOLOR for specific colors). Within the vertical stripe which is each player's display, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to 0 paints no color at all! That is, any 0 bit in a player stripe has no effect on the underlying playfield display.

Why call it a vertical stripe? Refer to Figure 5 for a rough idea of the player concept. If we define a shape within the bounds of this stripe (by changing some of the player's bits to ones as shown), we may move the player vertically by simply doing a circular shift on the contiguous memory block representing the player. Why is that easier than simply PLOTting something on the playfield and then moving it by PLOTting it again? First, since the player does not affect the playfield, any

pretty picture (or text or whatever) on the main screen remains unchanged. Second, because it's a lot easier to do a circular shift on a byte string than it is to change memory cells that are 40 (or 20 or 10?) bytes apart in memory.

Finally, the real clincher: even though vertical movement requires some shuffling of a string of bytes, horizontal movement is essentially effortless. Each and every player *and* missile has its own independent *register* (i.e., memory location) which controls its current horizontal position on the screen. Moving a player stripe horizontally is as easy as a single POKE from BASIC.

To summarize and simplify: A player is actually seen as a stripe on the screen eight pixels wide by 128 (or 256) pixels high. Within this stripe, the user may POKE or move bytes to establish what is essentially a tall, skinny picture (though much of the picture may consist of 0 bits, in which case the background "shows through"). Using a simple POKE, the programmer may then move this player to any horizontal location on the screen. To move a player vertically, though, one must do some sort of shift or move on the contents of the string of bytes displayed in the stripe.

From standard Atari BASIC, there is no easy way to move these stripes vertically (using a FOR/NEXT loop with PEEKs and POKEs is simply too slow). And while there now exist languages which have built-in mechanisms to do this movement (e.g., MOVE in Microsoft BASIC; MOVE and PMMOVE in BASIC A+), the overwhelming use of Atari BASIC has prompted many authors to try their hands at providing this movement in a form easily usable from Atari BASIC. This chapter includes sections detailing a few of the methods which have been worked out.

And now some final comments before we leave this introduction to player/missile graphics:

Missiles pretty much work just like players except that (1) they are only two bits wide instead of eight, (2) all four missiles share the same 128 or 256 bytes of memory (each using only its own bits in each byte), (3) each two-bit sub-stripe has an independent horizontal position register, and (4) by default a missile has the same color as its parent player. A later section in this chapter will delve a bit deeper into the mysteries of missiles.

There are essentially only five primary controls available to the P/M graphics user. We have already mentioned three: inde-

pendent control of the various player horizontal positions, independent control of the player's colors, and system-wide control over the resolution (128 bytes or 256 bytes per player, or simply "off").

In addition, each player and each missile has an independently controllable "width." A player or missile may be specified as single-width (narrow), double-width, or quadruple-width. This width does *not* affect the number of bytes or bits used for the display; it affects only the width of each individual pixel. Refer to Figure 6 for a diagram of a four-player system showing independent horizontal position and width control.

Incidentally, single-width players generated in the 128-byte vertical resolution mode have square pixels which are the same size as those in GRAPHICS 7, a presumably not altogether accidental happening.

The last control available to the user is the ability to specify where in memory the player and missile stripes are to be located. The rule is fairly simple: you need 2K bytes for single-line resolution (256 bytes per player), and it must be located on a 2K-byte memory boundary. For double-line resolution (128 bytes per player), you need a 1K-byte segment located on a 1K byte boundary.

Are you quick in arithmetic? How many 256-byte players can you put into 2K bytes? Or how many 128-byte players can 1K bytes hold? If you answered eight, you pass. If you answered five, you can go to the head of your Atari class. Indeed, with the Atari P/M memory map, you "waste" three players if you allocate the full amount of memory called for; see Figure 7 to see why.

Do you see the wasted memory? Does it need to be wasted? No. There is no reason why you can't put data, character sets, or what-have-you in this area. Indeed, in BASIC A+, part of the language is in this otherwise excess area.

And now you are ready to peruse the secrets unveiled herein; the darkest mysteries of player/missile graphics will become open to you. But don't be surprised if you find even more things that can be done with P/M graphics than we told you about here.

Figure 1. RAM considered as a linear "string" of bytes.

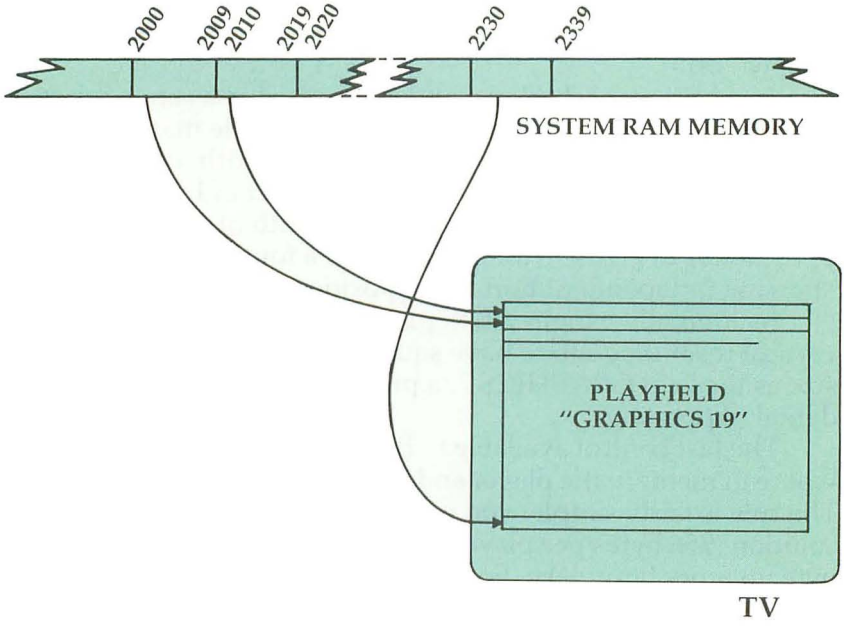


Figure 2. RAM considered as an array of 10-byte strings.

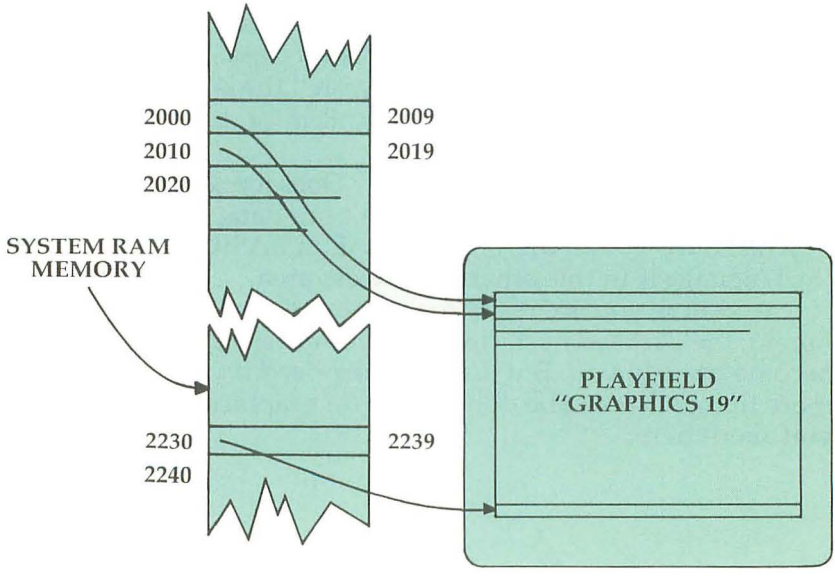


Figure 3. RAM considered as an array of 40-byte strings.

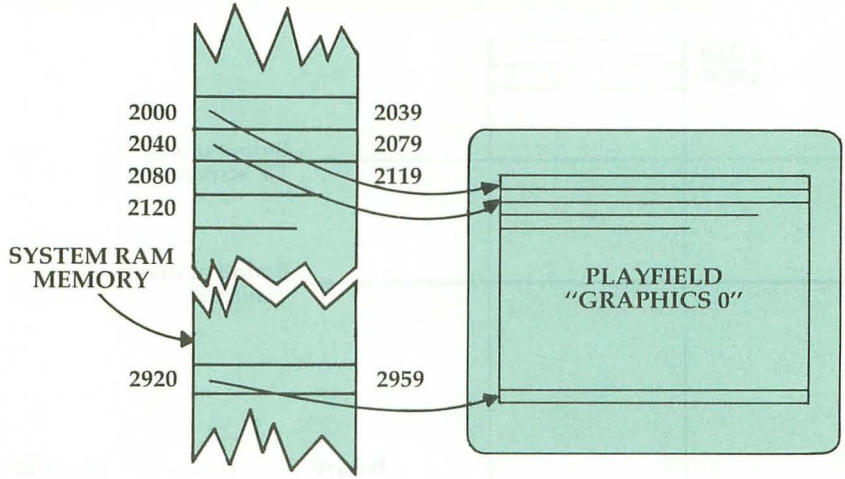


Figure 4. RAM considered two different ways for two different purposes.

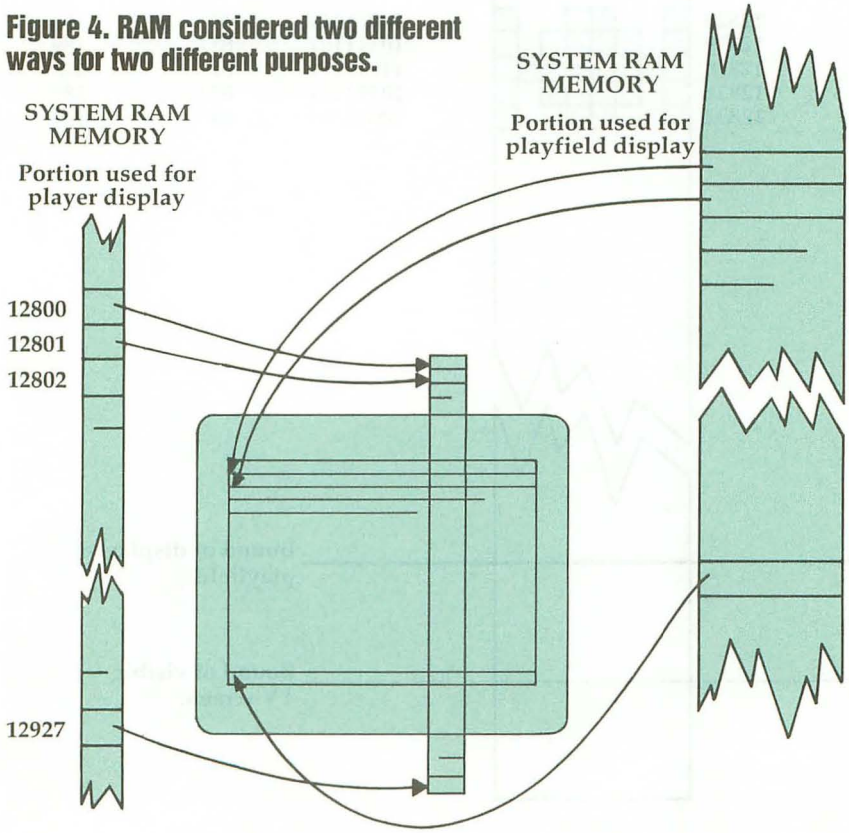


Figure 5. Detail – the display of Player Memory.

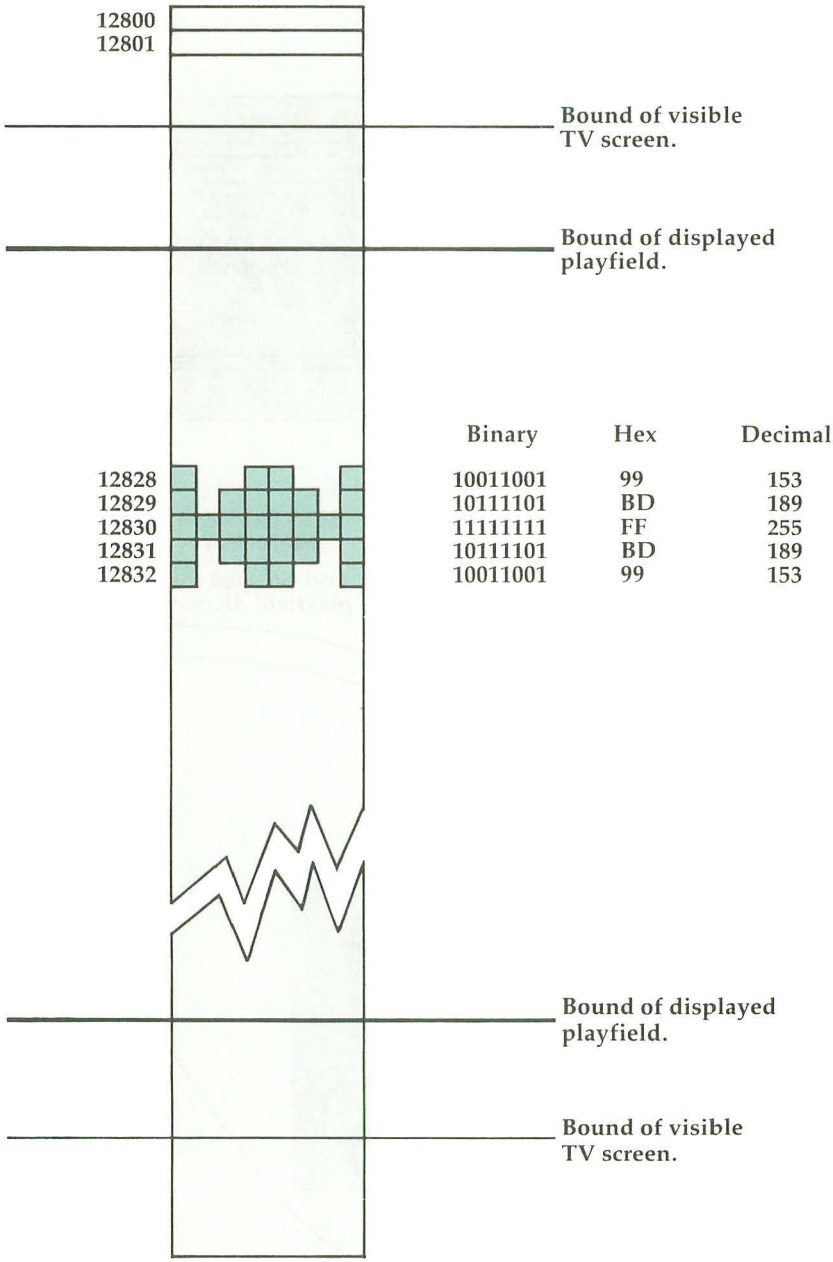


Figure 6. Four Players at once.
(Player 2 is double width and overlaps Player 3)

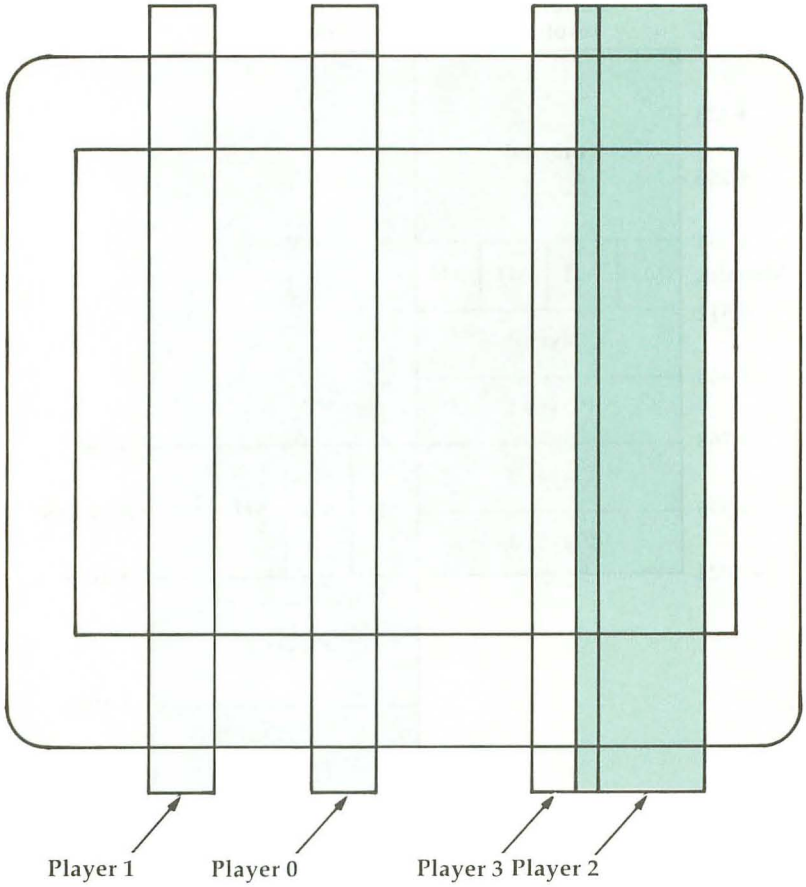


Figure 7. Player/missile graphics RAM positioning.

PMBASE must be on 1K boundary for double-line resolution, 2K boundary for single-line resolution.

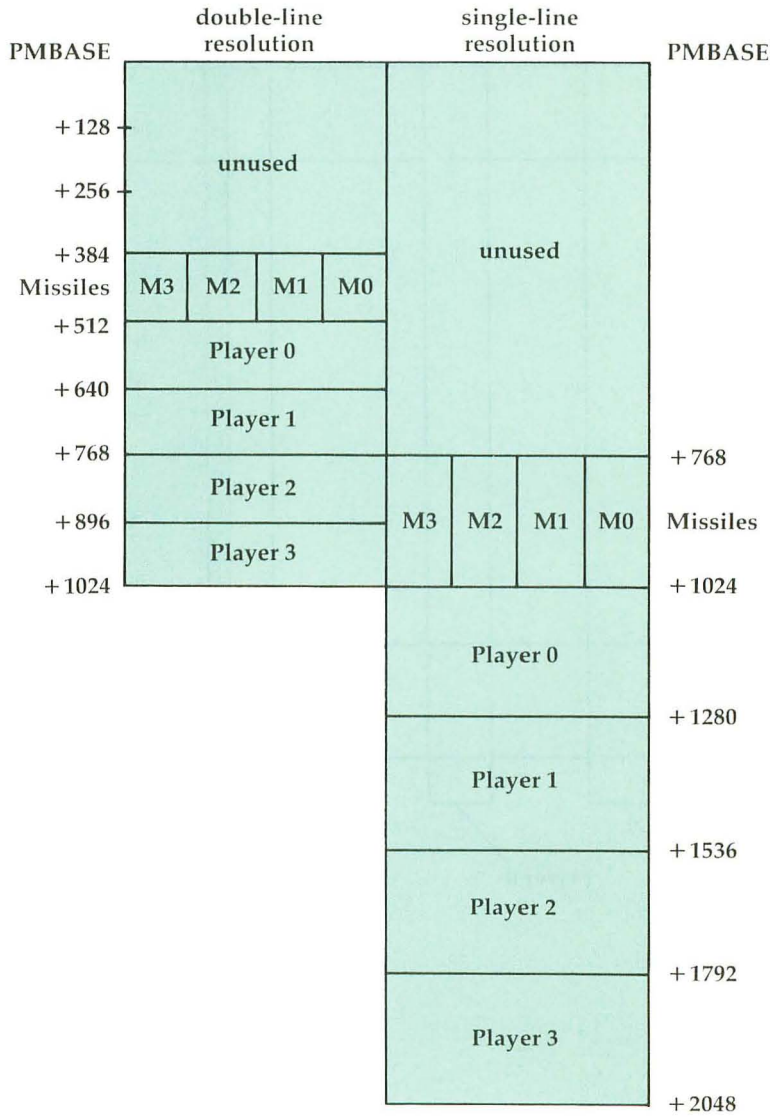


Figure 8. Important P/M memory locations.

Useful addresses
(all values in decimal)

559 put a 62 here for a single-line, a 46 for double-line resolution
623 sets player/playfield priorities (only one bit on!)
1: all players have priority over all playfield registers
4: all playfield registers have priority over all players
2: mixed. P0 & P1, then all playfield, then P2 & P3
8: mixed, PF0 & PF1, then all players, then PF2 & PF3
704 color of player/missile 0
705 color of player/missile 1
706 color of player/missile 2
707 color of player/missile 3
53248 horizontal position of player 0
53249 horizontal position of player 1
53250 horizontal position of player 2
53251 horizontal position of player 3
53252 horizontal position of missile 0
53253 horizontal position of missile 1
53254 horizontal position of missile 2
53255 horizontal position of missile 3
53256 size of player 0 (0 = normal, 1 = double, 3 = quadruple)
53257 size of player 1 (0 = normal, 1 = double, 3 = quadruple)
53258 size of player 2 (0 = normal, 1 = double, 3 = quadruple)
53259 size of player 3 (0 = normal, 1 = double, 3 = quadruple)
53277 A 3 here enables player/missile graphics, a 0 disables them.
54279 put high byte of PMBASE here

A Self-Modifying Player/Missile Graphics Utility

Kenneth Grace, Jr.

This excellent utility program removes much of the complexity from setting up player/missile graphics. It modifies itself into a skeleton program that can become the core of your game or graphics demo.

The utility in Program 1 sets up a skeleton program for Atari player/missile graphics. It presents a series of questions about the P/M situation you want to create and then modifies itself according to your responses. The resulting skeleton program includes some subroutines which you can use for controlling player and missile motion. They are based on string manipulations, so animation is also easy to accomplish.

I got the idea for this program after reading Bruce Frumker's article (**COMPUTE!**, August 1981, #15) on self-modifying programs. I hope this program will stimulate your thinking on other ways to use the self-modification capability built into the Atari.

There are several steps involved in setting up P/M graphics, and they have been covered in **COMPUTE!** and elsewhere. The steps are easy, but there are several choices available along the way (resolution, numbers of players and missiles, colors, initial positions, etc.). That's where this utility comes in.

It contains all the basic steps, and where there are choices to be made, they are presented to you. The program then uses Frumker's technique to add appropriate lines to the program. It also uses the same technique to delete lines that are not needed for your specific P/M setup, including the lines which ask the questions.

When the utility has finished, you are left with the skeleton of a P/M graphics program. You can LIST or RUN it at this point to check things out. But to make it a real program, you will have to draw the playfield and add the main loop for controlling motion, checking collisions, etc. In other words, the utility does just the P/M setup.

Since I make extensive use of Frumker's technique, I have split it into two subroutines, at 150 and 155. Between the two subroutine calls I put PRINT statements for the lines to be added to, or deleted from, the skeleton.

Aside from these two subroutines, the heart of the program is in lines 20-145. These lines present the series of questions through which you define your particular P/M arrangement. For example, lines 20-36 and 9020 take account of Fred Pinho's rules for placing P/M memory so that it doesn't overlap the memory for the BASIC GRAPHICS mode [*see his articles elsewhere in this book*]. The self-modifying feature is used after every question or two to add the appropriate statements to the setup section beginning at line 9000. At a few places the program STOPs while you enter DATA statements containing the bytes defining the shapes of the players and missiles.

Also scattered through this section are lines, such as 18, which delete the preceding lines, or delete the missile motion subroutines (when you have no missiles), or delete other unneeded lines in the section starting at 9000. If you have done any program editing on the Atari, you no doubt are aware of the keyboard "lock-up" problem. With all the deletions in this utility, I am almost inviting this disaster. Indeed, it cropped up many times as I was developing it.

Seemingly minor changes in the program would make the difference in whether it showed up or not. I say all this by way of warning. If you key in Program 1 exactly as shown, it should work OK. But if you decide to make improvements to it, you might run into the lock-up problem for certain combinations of inputs.

When the utility has finished running, you are left with lines 1, 159, appropriate subroutines from 160-198, a trivial loop at 200, and the P/M setup steps starting at 9000. Starting from this skeleton, I suggest that you use lines 2-158 for REMarks, opening titles, instructions, other subroutines, etc., and begin your main program at line 200. Additional setup steps, such as drawing the playfield, could go at the end of the section at 9000.

Motion Using Strings

I have included subroutines for player motion which use string manipulations. This method is described in George Blank's column in the April 1981 issue of *Creative Computing*. The basic idea is that you trick your Atari into treating the player/missile memory as the string array storage area for strings P0$, P1$, P2$, P3$, and M$. Lines 1 and 9500-9580 do this. You can then use Atari's fast string-handling routines for vertical motion or animation of the players.

In order for this to work, P0$,...,M$ must be the first variables mentioned in the program. You can assure this by turning off power momentarily and then typing line 1. In line 9500, VTAB is a pointer to the start of the variable table, which contains eight bytes for each variable. ATAB points to the start of the string array table, which is where the actual values are stored. Each pass through 9510-9580 modifies the eight bytes for Px$ (P0$, P1$, etc.) in the variable table, including the offset from ATAB where the actual values are stored (the P/M graphics memory).

The bytes defining the players are stored in strings D0$, D1$,... at lines 9090, 9140, etc. Each character in a string is stored in memory as a byte containing the corresponding ATASCII value. In this case, we want our data BYTE treated as though it were already an ATASCII value, so we use Dx$(I,I) = CHR$(BYTE). Note that this is a different way of using strings for P/M from Alan Watson's method (**COMPUTE!**, September 1981, #16). The demo (Program 2) mirrors Watson's example.

The descriptions Dx$ and DMx$ are initially read into P/M memory (i.e., into Px$ and M$) at lines 9600-9680. The string B$ is a "blanking" string; it is filled with ATASCII values of zero at line 9070.

I have included two subroutines for player motion. The routine at 160 handles vertical moves of one or two units. The strings Dx$ are set up to include two blanks (ATASCII zero) at the top and bottom of each player description. Thus, small vertical moves can be accomplished by writing Px$(Y(P) + DY) = Dx$. The blanks in Dx$ will make sure that the old image is wiped out. The variable P is a pointer to the player being moved (0, 1, 2, or 3); its value is set by your program before the subroutine is called, as are the position changes DX and DY. Incidentally, the array variables X(), Y(), and L() hold

the horizontal positions, vertical positions, and vertical lengths of the players. The corresponding variables for missiles are XM(), YM(), and LM().

The routine at 170 handles larger moves by blanking out the old player image with B$, changing the horizontal position to X(P) + DX, and rewriting the player image into Px$ at the new vertical position Y(P) + DY.

Vertical motion of a missile is slightly more difficult. The problem is that all four missiles are stored in the same memory block. Each missile occupies a two-bit slice of the eight-bit bytes in this memory block. Thus, we cannot simply write whole new bytes or blanks into this memory.

Instead, using a machine language routine, we do a logical AND of the existing memory with a binary mask, such as 11110011. This erases the old image in the appropriate two-bit slice, but leaves the rest of the missiles unchanged. Then we ADD the new image from DMx$. Since this image has zeros outside the two-bit slice, it won't affect the images of the other missiles. All of this is done at the vertical position of the new image. If there is a substantial vertical move involved, then two calls to the machine language routine are necessary: once to write B$ at the old position and once to write DMx$ at the new position.

Lines 9700-9740 read the machine language routine into the string MOVE$. The missile motion subroutine at 180 makes a USR call to this routine. The last variable in the USR call is the decimal equivalent of the binary mask. This subroutine assumes that vertical moves will be limited to one or two units (analogous to the player routine at 160). The subroutine at 190 handles the larger moves (analogous to the player subroutine at 170).

The Demo Program

Program 2 presents a demonstration of the use of the utility and motion routines. The demo attempts to duplicate Watson's animation program in **COMPUTE!** #16. The top part of the listing shows the answers you should give to the questions presented by the utility. The bottom part shows the lines to be added to the skeleton. Lines 300-530 match Watson's line numbers as closely as possible. A comparison of this demo with Watson's shows that the motion here is slightly faster – listen to the rate of the marching feet.

Finally, a word of caution: after keying in Program 1, save it on tape or disk before you run it. If you don't, you will find that a lot of your hard work has been wiped out.

Program 1. Player/Missile Graphics Utility.

```
1 DIM PO$(1),P1$(1),P2$(1),P3$(1),M$(
  1),X(3),Y(3),L(3),XM(3),YM(3),LM(3)
10 GRAPHICS 17:POSITION 2,3:? #6;"A S
   ELF-MODIFYING":POSITION 3,6:? #6;"
   PLAYER-MISSILE"
11 POSITION 2,9:? #6;"GRAPHICS UTILIT
   Y":POSITION 6,16:? #6;"ken grace"
12 FOR T=1 TO 2000:NEXT T
13 GRAPHICS 0:? :? "THIS UTILITY ASKS
    SEVERAL QUESTIONS{3 SPACES}ABOUT
   THE P-M GRAPHICS SITUATION YOU  WA
   NT TO SET UP."
14 ? :? "IT THEN MODIFIES ITSELF INTO
    A PROGRAMSKELETON.":? :? "SUBROUT
   INES FOR PLAYER AND MISSILE "
15 ? "MOTION ARE INCLUDED.":? :? "YOU
    ADD THE REST OF THE PROGRAM.":? :
   ? "ANIMATION IS POSSIBLE BY COPYIN
   G NEW "
16 ? "SHAPE STRINGS INTO THE STRINGS"
   :? "DEFINING THE PLAYERS.":? :? :?
    "PRESS START TO BEGIN."
17 X=PEEK(53279):IF X<>6 THEN 17
18 GOSUB 150:FOR I=10 TO 17:? I:NEXT
   I:GOSUB 155
20 ? CHR$(125):? :? "ENTER THE BASIC
   GRAPHICS MODE FOR THE PLAYFIELD":?
    "GR. ";:INPUT X
22 GOSUB 150:? "9000 GR. ";X:GOSUB 15
   5
24 ? "RESOLUTION DESIRED FOR PLAYERS:
   ":? "0 = DOUBLE-LINE":? "1 = SINGL
   E-LINE (FINER)":INPUT R
26 Y=INT(X/16):X=X-16*Y:IF X<=4 THEN
   S=8*(1+R)
28 IF X=5 THEN S=12+4*R
30 IF X=6 THEN S=16+8*R
32 IF X=7 THEN S=24+8*R
34 IF X=8 THEN S=36+4*R
36 GOSUB 150:? "9010 RES=";R;":S=";S:
```

```
   GOSUB 155:S=128*(1+R)
38 GOSUB 150:FOR I=18 TO 36 STEP 2:?
   I:NEXT I:GOSUB 155
40 ? "NUMBER OF PLAYERS TO BE DEFINED
   ";:INPUT NP
42 IF NP<4 THEN GOSUB 150:FOR I=NP TO
    3:? 9085+50*I:? 9090+50*I:NEXT I:
   GOSUB 155
44 IF NP<4 THEN GOSUB 150:FOR I=NP TO
    3:? 9600+10*I:NEXT I:GOSUB 155
48 GOSUB 150:FOR I=38 TO 44 STEP 2:?
   I:NEXT I:GOSUB 155
50 FOR I=0 TO NP-1
52 ? CHR$(125):? :? "COLOR (0 - 15) A
   ND INTENSITY (0 - 15) FOR PLAYER "
   ;I;:INPUT X,Y
54 GOSUB 150:? 9050+I;" POKE ";704+I;
   ",";16*X+Y:GOSUB 155
56 ? "WIDTH OF PLAYER ";I;":":? "0 =
   NORMAL":? "1 = TWICE NORMAL":? "3
   = FOUR TIMES NORMAL":INPUT X
58 GOSUB 150:? 9060+I;" POKE ";53256+
   I;",";X:GOSUB 155
60 ? "INITIAL HORIZONTAL POSITION (0
   - 255) FOR LEFT  EDGE OF PLAYER ";
   I;"  (45 TO 210ON SCREEN)";:INPUT
   X
62 GOSUB 150:? 9080+50*I;"X(";I;")=";
   X;":REM HORIZ POS OF PLAYER ";I:GO
   SUB 155
64 ? "VERTICAL LENGTH (BYTES) OF PLAY
   ER ";I;:INPUT X:? CHR$(125):?
66 ? "INITIAL VERTICAL POSITION OF TO
   P OF{3 SPACES}PLAYER  (1 TO ";S-X-
   1;")";:INPUT Y
68 GOSUB 150:? 9082+50*I;"Y(";I;")=";
   Y;":L(";I;")=";X+4;":REM VERT POS
   AND LENGTH":GOSUB 155
70 ? "USE LINES ";9100+50*I;" TO ";91
   20+50*I;" TO ENTER DATA  STATEMENT
   S WITH THE ";X;" BYTES DEFINING  P
```

```
   LAYER ";
72 ? :? "TYPE CONT WHEN FINISHED.":ST
   OP
74 NEXT I
78 GOSUB 150:FOR I=48 TO 74 STEP 2:?
   I:NEXT I:GOSUB 155
80 ? :? "HOW MANY MISSILES TO BE DEFI
   NED (0 TO 4)";:INPUT NM
82 IF NM=0 THEN GOSUB 150:FOR I=180 T
   O 188:? I:NEXT I:GOSUB 155
84 IF NM=0 THEN GOSUB 150:FOR I=190 T
   O 198:? I:NEXT I:GOSUB 155
86 IF NM<4 THEN GOSUB 150:FOR I=NM TO
    3:? 9285+50*I:? 9290+50*I:NEXT I:
   GOSUB 155
88 GOSUB 150:FOR I=78 TO 86 STEP 2:?
   I:NEXT I:GOSUB 155
90 IF NM=0 THEN ? CHR$(125):GOTO 119
92 S=0:FOR I=0 TO NM-1
94 ? CHR$(125):? :? "WIDTH OF MISSILE
    ";I:? "0 = NORMAL":? "1 = TWICE N
   ORMAL"
96 ? "3 = FOUR TIMES NORMAL":INPUT X:
   S=INT(4^I+0.1)*X+S
98 GOSUB 150:? "9064 POKE 53260,";S:G
   OSUB 155
100 ? "INITIAL HORIZONTAL POSITION OF
     MISSILE";I;:INPUT X
102 GOSUB 150:? 9280+50*I;"XM(";I;")=
    ";X;":REM MISSILE ";I;" HORIZ POS
    ":GOSUB 155
104 ? "VERTICAL LENGTH (BYTES) OF MIS
    SILE ";I:INPUT X:? CHR$(125)
106 ? :? "INITIAL VERTICAL POSITION O
    F TOP OF{3 SPACES}MISSILE (1 TO "
    ;128*(1+R)-X-1;")":INPUT Y
108 GOSUB 150:? 9282+50*I;" YM(";I;")
    =";Y;":LM(";I;")=";X+4;":REM VERT
     POS AND LENGTH":GOSUB 155
110 ? "USE LINE ";9300+50*I;" (TO ";9
    320+50*I;") TO ENTER DATA STATEME
```

```
    NTS WITH THE ";X;" 'BYTES' DEFINI
    NG"
112 ? "MISSILE ";I:X=INT(4^I+0.1):? :
    ? "ALLOWED VALUES ARE 0, ";X;", "
    ;2*X;", OR ";3*X:? :STOP
114 NEXT I
119 GOSUB 150:FOR I=88 TO 114 STEP 2:
    ? I:NEXT I:GOSUB 155
120 IF NM<4 THEN GOSUB 150:FOR I=NM T
    O 3:? 9650+10*I:NEXT I:GOSUB 155
125 IF NM=0 THEN GOSUB 150:FOR I=0 TO
     4:? 9700+10*I:NEXT I:GOSUB 155
129 GOSUB 150:? "119":? "120":? "125"
    :GOSUB 155
130 ? "PRIORITY SCHEDULE :":? :? "1 -
     PLAYERS 0-3,PLAYFLDS 0-3,BACKGND
    "
131 ? :? "2 - PLAYERS 0-1,PLAYFLDS 0-
    3,PLAYERS{6 SPACES}2-3,BACKGND"
132 ? :? "4 - PLAYFLDS 0-3,PLAYERS 0-
    3,BACKGND"
133 ? :? "8 - PLAYFLDS 0-1,PLAYERS 0-
    3,PLAYFLDS{5 SPACES}2-3,BACKGND"
134 ? :? "ALSO, THE NUMERICAL SUMS OF
     THE ABOVE CHOICES ARE ALLOWED, G
    IVING BLACK FOR OVERLAPS."
135 ? :? "ABOVE +32 GIVES COLOR IN OV
    ERLAPS":? :? "CHOICE"::INPUT X
136 GOSUB 150:? "9045 POKE 623,";X:GO
    SUB 155
137 ? :? "WHEN YOU SEE READY YOU MAY
    LIST OR RUN":FOR X=1 TO 900:NEXT
    X
140 GOSUB 150:FOR I=129 TO 137:? I:NE
    XT I:GOSUB 155
145 ? :? :? "140":? "145":? "150":? "
    155":? "156":? "POKE 842,12:? CHR
    $(125)":POSITION 0,0:POKE 842,13:
    STOP
150 SETCOLOR 1,9,4:? CHR$(125):? :RET
    URN
```

```
155 ? :? :? "CONT":POSITION 0,0:POKE
    842,13:STOP
156 POKE 842,12:? CHR$(125):? :SETCOL
    OR 1,9,10:RETURN
159 GOTO 9000
160 REM MOTION OF PLAYER P. X(P) AND
    Y(P) ARE X,Y POSITIONS. DX AND DY
     ARE CHANGES. USE FOR DY=-2,-1,0,
    1 OR 2.
161 TRAP 168:IF DY=0 THEN 167
162 ON P+1 GOTO 163,164,165,166
163 P0$(Y(P)+DY)=D0$:GOTO 167
164 P1$(Y(P)+DY)=D1$:GOTO 167
165 P2$(Y(P)+DY)=D2$:GOTO 167
166 P3$(Y(P)+DY)=D3$
167 POKE 53248+P,X(P)+DX:X(P)=X(P)+DX
    :Y(P)=Y(P)+DY:DX=0:DY=0:RETURN
168 DX=0:DY=0:GOTO 161
170 REM MOTION OF PLAYER P. USE FOR
    {3 SPACES}DY >2 OR <-2 (OR 0).
171 TRAP 177:ON P+1 GOTO 172,173,174,
    175
172 P0$=B$:POKE 53248,X(P)+DX:P0$(Y(P
    )+DY)=D0$:GOTO 176
173 P1$=B$:POKE 53249,X(P)+DX:P1$(Y(P
    )+DY)=D1$:GOTO 176
174 P2$=B$:POKE 53250,X(P)+DX:P2$(Y(P
    )+DY)=D2$:GOTO 176
175 P3$=B$:POKE 53251,X(P)+DX:P3$(Y(P
    )+DY)=D3$
176 X(P)=X(P)+DX:Y(P)=Y(P)+DY:DX=0:DY
    =0:RETURN
177 DX=0:DY=0:GOTO 171
180 REM MOTION OF MISSILE P. XM(P),YM
    (P)  ARE X,Y COORDS. DX,DY ARE CH
    ANGES.USE FOR DY=-2,-1,0,1 OR 2.
181 TRAP 158:IF YM(P)+DY<1 OR YM(P)+D
    Y+LM(P)>S OR DY=0 THEN DY=0:GOTO
    187
182 ON P+1 GOTO 183,184,185,186
183 Z=USR(MOVE,M+YM(P)+DY,DMO,LM(0),2
    52):GOTO 187
```

```
184 Z=USR(MOVE,M+YM(P)+DY,DM1,LM(1),2
    43):GOTO 187
185 Z=USR(MOVE,M+YM(P)+DY,DM2,LM(2),2
    07):GOTO 187
186 Z=USR(MOVE,M+YM(P)+DY,DM3,LM(3),6
    3)
187 POKE 53252+P,XM(P)+DX:XM(P)=XM(P)
    +DX:YM(P)=YM(P)+DY:DX=0:DY=0:RETU
    RN
188 DX=0:DY=0:POKE 53252+P,XM(P):RETU
    RN
190 REM MOTION OF MISSILE P. USE FOR
     DY>2 OR <-2  (OR 0).
191 TRAP 198:IF YM(P)+DY<1 OR YM(P)+D
    Y+LM(P)>S OR DY=0 THEN DY=0
192 ON P+1 GOTO 193,194,195,196
193 Z=USR(MOVE,M+YM(P),B,LM(P),252):P
    OKE 53252,XM(P)+DX:Z=USR(MOVE,M+Y
    M(P)+DY,DM0,LM(P),252):GOTO 197
194 Z=USR(MOVE,M+YM(P),B,LM(P),243):P
    OKE 53253,XM(P)+DX:Z=USR(MOVE,M+Y
    M(P)+DY,DM1,LM(P),243):GOTO 197
195 Z=USR(MOVE,M+YM(P),B,LM(P),207):P
    OKE 53254,XM(P)+DX:Z=USR(MOVE,M+Y
    M(P)+DY,DM2,LM(P),207):GOTO 197
196 Z=USR(MOVE,M+YM(P),B,LM(P),63):PO
    KE 53255,XM(P)+DX:Z=USR(MOVE,M+YM
    (P)+DY,DM3,LM(P),63)
197 YM(P)=YM(P)+DY:XM(P)=XM(P)+DX:DX=
    0:DY=0:RETURN
198 DX=0:DY=0:GOTO 191
200 GOTO 200
9015 POKE 559,46+16*RES
9020 PMBASE=PEEK(106)-S:POKE 54279,PM
     BASE:PMBASE=PMBASE*256
9030 POKE 53277,3:S=128:IF RES=1 THEN
      S=255
9070 DIM B$(S):B=ADR(B$):B$(1)=CHR$(0
     ):B$(S)=CHR$(0):B$(2)=B$
9085 DIM DO$(L(0)):DO$=B$(1,L(0)):POK
     E 53248,X(0)
```

```
9090 RESTORE 9100:FOR I=3 TO L(0)-2:R
     EAD BYTE:D0$(I,I)=CHR$(BYTE):NEX
     T I
9135 DIM D1$(L(1)):D1$=B$(1,L(1)):POK
     E 53249,X(1)
9140 RESTORE 9150:FOR I=3 TO L(1)-2:R
     EAD BYTE:D1$(I,I)=CHR$(BYTE):NEX
     T I
9185 DIM D2$(L(2)):D2$=B$(1,L(2)):POK
     E 53250,X(2)
9190 RESTORE 9200:FOR I=3 TO L(2)-2:R
     EAD BYTE:D2$(I,I)=CHR$(BYTE):NEX
     T I
9235 DIM D3$(L(3)):D3$=B$(1,L(3)):POK
     E 53251,X(3)
9240 RESTORE 9250:FOR I=3 TO L(3)-2:R
     EAD BYTE:D3$(I,I)=CHR$(BYTE):NEX
     T I
9285 DIM DM0$(LM(0)):DM0$=B$(1,LM(0))
     :POKE 53252,XM(0)
9290 RESTORE 9300:FOR I=3 TO LM(0)-2:
     READ BYTE:DM0$(I,I)=CHR$(BYTE):N
     EXT I:DM0=ADR(DM0$)
9335 DIM DM1$(LM(1)):DM1$=B$(1.LM(1))
     :POKE 53253,XM(1)
9340 RESTORE 9350:FOR I=3 TO LM(1)-2:
     READ BYTE:DM1$(I,I)=CHR$(BYTE):N
     EXT I:DM1=ADR(DM1$)
9385 DIM DM2$(LM(2)):DM2$=B$(1,LM(2))
     :POKE 53254,XM(2)
9390 RESTORE 9400:FOR I=3 TO LM(2)-2:
     READ BYTE:DM2$(I,I)=CHR$(BYTE):N
     EXT I:DM2=ADR(DM2$)
9435 DIM DM3$(LM(3)):DM3$=B$(1,LM(3))
     :POKE 53255,XM(3)
9440 RESTORE 9450:FOR I=3 TO LM(3)-2:
     READ BYTE:DM3$(I,I)=CHR$(BYTE):N
     EXT I:DM3=ADR(DM3$)
9500 VTAB=PEEK(134)+256*PEEK(135):ATA
     B=PEEK(140)+256*PEEK(141)
9505 OFFSET=PMBASE+512*(1+RES)-ATAB
```

```
9510 FOR I=0 TO 4
9520 V3=INT(OFFSET/256):V2=OFFSET-256
     *V3
9530 POKE VTAB+2,V2:POKE VTAB+3,V3
9540 POKE VTAB+4,128*(1-RES):POKE VTA
     B+5,RES
9550 POKE VTAB+6,128*(1-RES):POKE VTA
     B+7,RES
9560 VTAB=VTAB+8:OFFSET=OFFSET+128*(1
     +RES)
9570 IF I=3 THEN OFFSET=PMBASE+384*(1
     +RES)-ATAB
9580 NEXT I
9600 P0$=B$:P0$(Y(0))=D0$
9610 P1$=B$:P1$(Y(1))=D1$
9620 P2$=B$:P2$(Y(2))=D2$
9630 P3$=B$:P3$(Y(3))=D3$
9650 M$=B$:M$(YM(0))=DM0$:M$(YM(0)+LM
     (0))=B$
9660 FOR I=1 TO LM(1):J=YM(1)+I-1:M$(
     J,J)=CHR$(ASC(M$(J,J))+ASC(DM1$(
     I,I))):NEXT I
9670 FOR I=1 TO LM(2):J=YM(2)+I-1:M$(
     J,J)=CHR$(ASC(M$(J,J))+ASC(DM2$(
     I,I))):NEXT I
9680 FOR I=1 TO LM(3):J=YM(3)+I-1:M$(
     J,J)=CHR$(ASC(M$(J,J))+ASC(DM3$(
     I,I))):NEXT I
9700 DIM MOVE$(38):MOVE=ADR(MOVE$):M=
     ADR(M$)-1
9710 RESTORE 9730
9720 FOR I=1 TO 37:READ BYTE:MOVE$(I,
     I)=CHR$(BYTE):NEXT I
9730 DATA 104,104,133,204,104,133,203
     ,104,133,206,104,133,205,104,104
     ,133,207,104,104,133,208
9740 DATA 160,0,177,203,37,208,113,20
     5,145,203,200,196,207,208,243,96
9999 GOTO 200
```

Program 2. Animation Demo.

RUN the utility in Program 1 and give the following answers:

Graphics Mode: 18
Resolution: 0
Number of Players: 1
Color, Intensity: 1,6
Width: 0
Horizontal Position: 127
Length: 9
Vertical Position: 63

9100 DATA 126,90,66,60,219,189,102,102,231
CONT

Number of Missiles: 0
Priority: 1

Then add the following lines to the skeleton program:

```
200 DIM DO1$(13),DO2$(13),DO3$(13)
210 DO1$=DO$:DO2$=B$:DO3$=B$:P=0
220 RESTORE 520
230 FOR I=3 TO 11:READ BYTE:DO2$(I,I)
    =CHR$(BYTE):NEXT I
240 FOR I=3 TO 11:READ BYTE:DO3$(I,I)
    =CHR$(BYTE):NEXT I
250 SETCOLOR 4,7,2
300 REM ***VIEW POINTER & STRING ***
310 C=C+1
320 IF C>4 THEN C=1
330 ON C GOTO 340,350,340,360
340 DO$=DO1$:GOTO 370
350 DO$=DO2$:GOTO 370
360 DO$=DO3$
370 PO$(Y(0))=DO$
380 FOR I=1 TO 9
385 IF C=2 OR C=4 THEN SOUND 0,28*I,6
    ,9-I
390 NEXT I
400 REM *** MOTION ROUTINE ***
410 A=STICK(0)
420 IF A=15 THEN 310
430 IF A=11 THEN X(0)=X(0)-1:POKE 532
    48,X(0)
```

```
440 IF A=7 THEN X(0)=X(0)+1:POKE 5324
    8,X(0)
450 IF A=13 THEN DY=1:GOSUB 160
460 IF A=14 THEN DY=-1:GOSUB 160
470 GOTO 310
520 DATA 126,90,66,60,219,189,102,230
    ,7
530 DATA 126,90,66,60,219,189,102,103
    ,224
```

Adding High-Speed Vertical Positioning To P/M Graphics

David H. Markley

Although fast horizontal movement of players and missiles is easy with BASIC, vertical movement is much slower. This article provides a machine language routine which can be attached to a BASIC program to speed things up considerably.

By now many of you have been experimenting with programs incorporating the advanced player/missile graphics of the Atari. As you may have observed, player images can be moved horizontally across the playfield quite easily just by placing the player's horizontal coordinate (0-120) into its associated horizontal position register. Vertical positioning with P/M graphics, however, is somewhat more difficult. Since the player's vertical position on the playfield inversely corresponds to its position within the image memory, it is necessary to relocate each byte of the image up or down within the memory to produce vertical movement. For example, if we move the player's image to higher address locations within the image memory, the player will appear to move downward on the playfield.

A BASIC routine can be written using PEEKs and POKEs to move the player within the image memory, but for most applications this method is too slow. An alternative, however, is to use a small, general purpose vertical positioning routine written in 6502 machine language which can be called by BASIC's USR instruction.

The vertical positioning routine shown in Program 1 is relatively simple, but provides the user with a flexible and easy method of handling P/M graphics within a BASIC program. This not only provides a valuable tool to use with player/missile graphics, but for those of you who have not used machine

language routines with BASIC, it will also provide some insight into this area. The routine is called by a BASIC statement similar to:

```
DUMMY=USR(VP,IMAGE,LAST LOCATION, NEW
    LOCATION)
```

The variable to the left of the equal sign, called "DUMMY", is used by some machine language subroutines as a target for a value returned to the program. The vertical positioning routine, however, does not return a usable value, but the DUMMY variable is still required to satisfy Atari's USR format requirements. Any variable may be used in place of DUMMY. Within the parentheses of the command are four arguments. The first argument, VP, is the transfer address to the Vertical Positioning routine which has been placed into a free area of memory. Loading of the VP routine into memory will be described later with a program application example. Following the transfer address argument (which, by the way, is also required for any USR routine called by BASIC) are three arguments which are passed to the VP routine.

These arguments are the address of the image's data structure, the address of the image's current position in the P/M image memory, and the address of its new position (the destination). As usual in P/M graphics, each image requires a small data structure. This provides the VP routine with a pattern of the actual image which it will vertically reposition. An example of a typical image data structure is shown in the figure. This is identical to the usual way in which images are drawn in P/M graphics, except for one additional byte which must be tacked onto the front of the data. The first byte of data provides the VP routine with the image's size in bytes. The second and following bytes are used to form a bit map pattern of the image as it would appear in the P/M image memory.

The next two arguments contained in the USR command tell the VP routine the image's current and new positions. These arguments are actual addresses into the image memory; therefore, care must be taken to assure that they do not access another area of memory by mistake.

Routine Operation

The program begins with an initialization step in which the three arguments passed to it by the USR command are removed from the processor's stack and placed into an area in page zero

where they can be more easily used. You may have noticed that a total of seven bytes are popped off the stack during this operation. This is because the USR command always places a one-byte argument count onto the stack, followed by the arguments themselves. The arguments are always two bytes in length.

Once the initialization task is complete, the routine is ready to begin its intended task of moving the player image. Basically, the operation is performed in two steps. The image data is first removed from its current location and then copied to its new location. Before either step can be executed, the routine must first look at the image's data structure and get the image size parameter. This value tells the routine how large an image it must handle and thus determines the number of bytes it must remove and restore. To remove an image from its current location, the routine simply goes to the current location address and writes zeros into an X number of memory locations indicated by the size parameter. Replacement of the image is done by copying from the image's data structure an X number of bytes, also determined by the size parameter, to the image memory starting at the address specified by the new position argument.

In some cases it may not be desirable to have the VP routine perform both the delete and restore functions. One example would be if the player image is to be removed from the screen and not restored at a new location. This can be handled by using the following routine call:

```
DUMMY=USR(VP,IMAGE,CURRENT LOCATION,0)
```

The zero in the new location argument tells the VP routine not to attempt to restore the image. Likewise, the delete function can be disabled by placing a zero in the current location argument.

Let's Have Some Fun

Now that we have looked at the Player/Missile Vertical Positioner routine, let's put it to work. The following game will show you how to load the player images and VP routine into memory and how to use the routine in other ways besides vertical positioning.

This game, which I call "Island Jumper," involves the cooperation of two characters named Crash Coleman and Deadeye Dan. Crash is the pilot of a reliable (but not so stable)

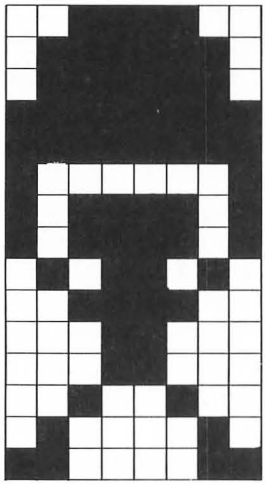
airplane, the "Leaping Lucy." Crash has had only one flying lesson, but has courageously volunteered to make this flight so that you can see the VP routine in action. Although he has successfully managed to get the Leaping Lucy off the ground, he seems to be having some trouble keeping her in level flight. Our other daredevil of the sky, Deadeye Dan, will attempt, with your help, to jump out of Crash's airplane and land on Target Island. Since the ground seems to be a bit unstable from Dan's point of view, he is having difficulty figuring out when to jump and asks that you help him by pulling back on your joystick controller when you think he's on target.

Dan will make a total of five jumps each time you play the game. He will try to land on top of a sand dune on the left side of the island. If he makes the jump on Crash's first pass over the island and lands on the dune with both feet, you get 30 points. If you don't give Dan the signal to jump during the first pass, Crash will continue to fly over the island until a jump is made. Each additional pass will deduct eight points from Dan's maximum obtainable score.

Dan can also land in the area between the sand dune and the palm tree, but you will receive a maximum of 15 points for the jump. At the completion of the game, the computer will give you both a final score for the last game played and the highest score for all games played since the last RUN command was entered. To play another game, press the button on the joystick controller.

The data for the VP routine and the player data structures is read from data statements and POKEd into memory by lines 110 through 310 of the program. It is loaded into memory page six (starting at address 1536), which is a 256-byte area in memory that Atari has reserved for user binary data and machine language routines. Once the data structures and VP routine are loaded into memory, they are referenced in the BASIC program by variable names whose values have been set to the starting address of the data structure or VP routine they represent.

Figure . Image Data Structure for the Player/Missile Vertical Positioner Routine.



Byte Number	Byte Value
1	60
2	126
3	126
4	255
5	255
6	129
7	189
8	189
9	90
10	60
11	24
12	24
13	36
14	66
15	195

DATA 15,60,126,126,255,255,129,189,189,90,60,24,24,36,66,195

Program 1. Island Jumper.

```
10 REM VERTICAL POSITIONER EXAMPLE
20 REM "ISLAND JUMPER"
50 GRAPHICS 2:POKE 752,1
60 SETCOLOR 4,9,4
70 ? #6:? #6:? #6:? #6;"{6 SPACES}ISL
   AND"
80 ? #6:? #6;"{6 SPACES}JUMPER"
90 ? ,"{6 SPACES}BY"
100 ? :? ,"DAVID MARKLEY"
110 VP=1536
120 FOR G=0 TO 93
125 READ D
130 POKE VP+G,D
135 NEXT G
140 REM ** VERTICAL POSITIONER CODE *
    *
150 DATA 104,162,5,104,149,220,202,16
    ,250,198,220,198,222,160,0,177,22
    4,170
160 DATA 168,165,223,240,9,169,0,145,
    222,136,208,249,138,168,165,221,2
    40,7,177,224,145,220,136,208,249,
    96
170 REM ** AIRPLANE DATA **
180 APIMG=VP+44
190 DATA 6,142,132,255,255,4,14
200 REM ** JUMPER DATA **
210 JPIMG=APIMG+7
220 DATA 9,189,189,90,60,24,24,36,66,
    129
230 REM ** JUMPER & CHUTE DATA **
240 JSIMG=JPIMG+10
250 DATA 15,60,126,126,255,255,129,18
    9,189,90,60,24,24,36,66,195
260 REM ** WAVING JUMPER **
270 JWIMG=JSIMG+16
280 DATA 15,0,0,0,0,0,128,188,188,88,
    60,26,25,37,66,195
290 REM ** DATA USED TO CLEAR MEMORY
    **
```

```
300 CLEAR=JWIMG+16
310 DATA 255
320 FOR D=1 TO 300:NEXT D
330 GRAPHICS 5
340 SETCOLOR 2,9,2
350 SETCOLOR 4,8,6
360 I=PEEK(106)
365 X=I*256-1172
370 POKE X,112
371 POKE X+1,71
372 POKE X+2,96
373 POKE X+3,I-1
374 POKE X+4,112
375 POKE X+5,74
376 POKE X+6,160
377 POKE X+7,I-5
380 I=I-8
390 POKE 54279,I
400 J=I*256+513
410 POKE 559,46
420 POKE 53256,1
430 POKE 53277,3
440 POKE 704,56
450 POKE 705,12
460 D=USR(VP,CLEAR,J,0)
465 SLOPE=2
470 TOP=J+17
480 BOT=J+55
490 SETCOLOR 0,12,8
500 SETCOLOR 1,1,2
510 COLOR 2
520 PLOT 37,34:DRAWTO 42,34
530 PLOT 36,35:DRAWTO 49,35
540 PLOT 47,29:DRAWTO 47,34
550 COLOR 1
560 PLOT 43,30:DRAWTO 47,27
570 PLOT 51,30:DRAWTO 47,27
580 PLOT 47,27:DRAWTO 49,30
590 PLOT 47,27:DRAWTO 45,30
600 PLOT 46,27
610 HSCORE=0
620 LAPOS=0
```

```
630 APOS=J+70
640 I=-1
650 JUMP=5
660 SCORE=0
670 PNTS=30
680 JMP=0
690 SOUND 0,31,4,4
700 POKE 623,4
710 JSTOP=J+219
720 FOR G=20 TO 245 STEP 3
730 POKE 53248,G
740 D=USR(VP,APIMG,LAPOS,APOS)
750 IF JMP=0 AND G<180 AND STICK(0)<>
    15 THEN JMP=APOS+132:POKE 53249,G
    +4:IMG=JPIMG:D=USR(VP,IMG,0,JMP)
760 LJMP=JMP
770 IF JMP=0 THEN 880
780 JMP=JMP+3
790 IF JMP<J+200 THEN HJMP=G+4:POKE 5
    3249,HJMP:SOUND 1,G,10,8:GOTO 860
800 IMG=JSIMG
804 JMP=JMP-2
808 SOUND 1,0,0,0
810 IF HJMP>=122 AND HJMP<=126 THEN J
    STOP=J+208:GOTO 860
820 IF HJMP<120 OR HJMP>134 THEN 860
830 JSTOP=J+210
840 POKE 623,1
850 IF PNTS>15 THEN PNTS=15
860 IF JMP>JSTOP THEN 940
870 D=USR(VP,IMG,LJMP,JMP)
880 LAPOS=APOS
890 APOS=APOS+I
900 D=USR(VP,APIMG,LAPOS,APOS)
910 IF APOS>BOT THEN I=-SLOPE
920 IF APOS<TOP THEN I=SLOPE
930 NEXT G
940 IF JMP<J AND PNTS>9 THEN PNTS=PNT
    S-8:GOTO 1220
950 IF JMP<J THEN 1220
970 IF HJMP<120 OR HJMP>134 THEN TONE
    =8:GOTO 1010
```

```
980 SCORE=SCORE+PNTS
985 TONE=12
990 D=USR(VP,JWIMG,0,JMP-1)
1000 ? "SCORE ";SCORE:? :?
1010 FOR D=15 TO 0 STEP -1
1020 SOUND 1,12,TONE,D
1030 FOR I=1 TO 10:NEXT I
1040 NEXT D
1050 SOUND 0,0,0,0
1055 SOUND 1,0,0,0
1060 JUMP=JUMP-1
1070 IF JUMP<>0 THEN 1170
1080 IF SCORE>HSCORE THEN HSCORE=SCOR
     E
1090 FOR I=1 TO 120
1100 IF I=1 THEN ? "HIGH SCORE ";HSCO
     RE:? :?
1110 IF I=60 THEN ? "FINAL SCORE ";SC
     ORE:? :?
1120 IF STRIG(0)=1 THEN 1150
1130 D=USR(VP,CLEAR,J,0)
1135 PRINT
1140 GOTO 630
1150 NEXT I
1160 GOTO 1090
1170 ? "JUMP ";6-JUMP:? :?
1180 FOR D=0 TO 250:NEXT D:?
1190 D=USR(VP,CLEAR,J,0)
1195 I=SLOPE
1200 IF RND(0)>0.5 THEN I=-SLOPE
1210 GOTO 670
1220 POKE 77,0
1225 GOTO 690
1230 END
```

Program 2. Assembly language representation of the P/M Vertical Positioner Routine.

```
10 ;P/M VERTICAL POSITIONER
20 NEW=220
30 CURRENT=222
40 IMAGE=224
50 START   PLA              ;REMOVE ARGUMENT
                            BYTE COUNT
60         LDX#5            ;REMOVE 6 BYTES
70  LP1    PLA              ;AND PLACE IN PAGE ZERO
80         STA NEW,X
90         DEX
100        BPL LP1
110        DEC NEW
120        DEC LAST
130        LDY#0
140        LDA (IMAGE),Y    ;GET IMAGE BYTE COUNT
150        TAX
160        TAY
170        LDA LAST+1       ;IF ZERO DON'T DELETE
180        BEQ SKIPD
190 LP2    LDA#0            ;DELETE IMAGE
200        STA (LAST),Y
210        DEY
220        BNE LP2
230 SKIPD  TXA
240        TAY
250        LDA NEW+1        ;IF ZERO DON'T RESTORE
260        BEQ SKIPR
270 LP3    LDA (IMAGE),Y    ;COPY IMAGE DATA TO NEW
280        STA (NEW),Y      ;ADDRESS
290        DEY
300        BNE LP3
310 SKIPR  RTS
```

P/M Graphics Made Easy

Tom Sak and Sid Meier

There's more than one way to obtain fast vertical movement with player/missile graphics. This article, and the one following, shows how vertical blank interrupts may be used for this purpose.

Many people have called the Atari's graphics capabilities its best feature, especially the player/missile graphics. We won't argue, but how many of you have backed away because it looks too difficult to handle in BASIC or you simply are not satisfied with the execution speeds which you are able to achieve?

Well, no more excuses! We've got a machine language subroutine that you can use with BASIC to achieve exciting graphics performance without a lot of muss and fuss. As a matter of fact, you make only one setup call to the subroutine and then forget it! And we promise you need know nothing about machine language. Just a few POKEs and you'll have your players dancing around the television screen.

You Don't Need To Know Machine Language

There have been a number of very helpful articles published describing the essential player/missile graphics information. We're going to assume that you are familiar with the fundamentals, but we'll review highlights as they're required.

A feature of the Atari with which you may not be familiar is its "interrupt" mechanism and how you can let it move your players for you at machine language speed – without the overhead of calling it from your BASIC program. Before we explore this useful feature, let's take a quick refresher course on interrupts.

As you know, the Atari keeps itself pretty busy doing its "housekeeping" chores even while it is interpreting your BASIC program. Among other things, the Atari must maintain the steady delivery of information to your television set, allowing

it to paint a constantly up-to-date picture of the display data. Multiple, concurrent activities are performed by allowing one particular activity to periodically interrupt another.

The traditional analogy is that of a busy business executive who, while engaged in a meeting with an associate, is interrupted by a telephone call. The ringing phone signals the interrupt; the executive "checkpoints" his meeting and answers the phone. After disposing of the call, the executive resumes his meeting at the point of interruption.

A similar circumstance occurs each time a complete picture is painted by your television set. The television's electron beam paints the picture by sweeping horizontal rows across the picture tube beginning in the upper left-hand corner and ending in the lower right. The beam is turned off when it reaches the lower right corner and is returned to its upper left starting position. This return trip is essentially a vertical positioning movement, so this period when the beam is turned off is known as the *vertical blank time*.

Move During Vertical Blanks

The onset of the vertical blank cycle serves as an opportunity for the Atari's ANTIC chip to signal an interrupt, the vertical blank or VBLANK interrupt. The operating system uses this occasion to perform some of its "housekeeping" duties. Fortunately, the operating system designers allow us to include a machine language subroutine which can be executed as one of these tasks.

The machine language vertical blank interrupt player movement subroutine described here is called VBLANK PM, and it allows you to simply POKE the next x and y coordinate at which your player is to be displayed. There is no need to repeatedly call the subroutine from BASIC via the USR function. The subroutine will be automatically executed during the next vertical blank period. It is possible to move the players every time a new screen is painted on the television – and that's 60 times a second!

You may recall from other articles that an appropriate POKE to location 53248 (and the three memory locations following) permits you to position players zero through three horizontally along the x-axis. It's not quite as easy to position the players vertically along the y-axis. Not until now!

The VBLANK PM subroutine takes care to move the players in both directions. Movements along the vertical axis involve "erasing" and rewriting the player in the new position. VBLANK PM does this for you, automatically. There are a few things which you must do for VBLANK PM, however.

First, you must get the VBLANK PM machine language subroutine into memory and notify the operating system that it is to be included as one of the "housekeeping" tasks to be performed as a part of servicing the vertical blank interrupt. Next, it's up to you to draw your players and tell VBLANK PM how tall they are. After initialization, VBLANK PM looks after the positioning of your players until either a warm start (pressing SYSTEM RESET) or a cold start (power-off, power-on sequence) is performed.

The program is an example of the initialization and use of the VBLANK PM subroutine. This program causes VBLANK PM to be loaded and initialized and players zero and one to be drawn and then moved about the television screen in a random pattern. The players are male and female gender symbols which the program "dances" around the screen.

Lines 100 through 200 are the main program; we'll save an explanation of these lines until after you've gained some insight into the initialization subprogram contained in lines 1000 through 1110. The VBLANK PM machine language subroutine is expressed in the DATA statements numbered 2000 through 2100. Finally, lines 3000 through 3020 supply a description of the two players used in this example.

The first task is to load VBLANK PM into page six of memory. Page six is locations 1536 through 1791 (hexadecimal 600 through 6FF) and has been left available by Atari's software designers for applications such as this one. These 256 bytes of memory are not disturbed by BASIC or DOS; however, a cold start does cause page six to be cleared to zeros. Line 1010 causes the VBLANK PM to be read and POKEd into memory. Line 1020 clears a few locations used by the subroutine; this statement can be omitted if you are sure that page six has not been altered since the last cold start.

We're going to employ the Atari's ANTIC chip direct memory access (DMA) facility to transfer graphics information from memory to the television using single-line resolution. (You might want to re-read the introductory article of this chapter or just "trust us on this one!") This means that we must allocate

2K (2048) bytes of memory for the storage of players. In line 1030 we obtain the page number of RAMTOP, deduct 16 pages, and call the result the base of the required 2K byte allocation.

Memory Allocation

Why 16 pages? Well, first consider that 2K bytes are eight pages (a page contains 256 bytes) and that, depending on the graphics mode (i.e., GRAPHICS 0 through GRAPHICS 8), you must allow sufficient space at the top of RAM to contain the display list and screen data. Incidentally, the player/missile 2K byte allocation must begin at an address which is a multiple of 2048; we call this starting address PMBASE.

One more cautionary note: you will have to allow more than 16 pages between PMBASE and RAMTOP if you are using graphics modes six through eight. Articles elsewhere in this book provide greater detail in this area.

The figure depicts the 2K byte memory allocation. Remember, we didn't design this scheme, Atari did, and we're not sure why, but there is a considerable amount of unused space involved. You can use the lower, unused bytes for your own purposes without disturbing anything, if you like. We're going to use only the upper 1K bytes.

Player zero occupies PMBASE + 1024 through PMBASE + 1279; player one is situated in locations PMBASE + 1280 through PMBASE + 1535, and so on for players two and three. Line 1040 clears any residual data – if you're in a hurry and are sure that this area is already clear (i.e., following a cold start), you won't need line 1040.

Memory Allocation For Single-Line Resolution of P/M Graphics.

Lines 1050 and 1060 are used to draw players zero and one. VBLANK PM expects the players to be drawn so that their top line is initially placed at the beginning of the individual player's storage area. The player can be as tall as you like up to 255 lines; of course, you will never see all of a player which is that tall on the screen at the same time.

Next you can see that we've taken advantage of the Atari's special memory locations for some functions. You establish the players' colors with a POKE into locations 704 through 707 for players zero through three, respectively. Line 1070 is used to set the colors and assumes that you've set the variables PCOL0, PCOL1, PCOL2, and PCOL3 already.

Line 1080 establishes the positioning addresses which you will be using later to signal player movements using only POKEs. PLX and PLY are the locations POKEd to establish the next x and y position of player zero. A POKE into location PLX+1 and PLY+1 accomplishes the same thing for player one, and so forth for players two and three. PLL (and PLL+1, PLL+2, and PLL+3) are POKEd to inform VBLANK PM of the length (or height) of each player.

Line 1090 initializes the remaining control parameters. A 62 is POKEd into location 559 to set the single line player/missile resolution graphics; a one placed into location 623 establishes the player/playfield priorities, giving the players priority over the playfield. (You can change this to suit your purposes, if you wish.) Location 1788 is in VBLANK PM and is POKEd with the number of the first page containing player/missile data. Locations 53277 and 54279 are used to switch on the DMA graphics data transfer facility and to tell the ANTIC chip where in memory to find the player graphics data.

Wrapping Up The Loose Ends

You're almost ready to go! A subroutine call to VBLANK PM from line 1100 allows VBLANK PM to notify the operating system of both its presence and its desire to be automatically invoked as a part of the vertical blank interrupt process. This is the only time in which your BASIC program must explicitly call VBLANK PM.

Okay, to wrap up loose ends, let's take a quick look at the main program – lines 100 through 200. Line 100 turns off the cursor, clears the screen, and provides a black background so that we can readily see the players.

Line 110 sets the players' colors before the VBLANK PM initialization subprogram is executed. You know how to set the colors, right? Multiply the color number by 16 and add the desired intensity – the color and intensity numbers are the same as those used in the SETCOLOR command. Line 120 assures that VBLANK PM is launched.

Line 130 illustrates the manner in which you pass instructions to VBLANK PM. Here we are telling VBLANK PM that both players are eight lines tall. You can change this parameter at any time – we have a little surprise for you later about why you might want to change this parameter.

Lines 140 and 150 establish the initial television screen positions of players zero and one, respectively. A word about the available values for the x and y coordinates might be helpful, since not all x and y values will result in the player being displayed. There are 255 x positions, with only 160 of these appearing across the television screen beginning with an x value of 48.

Similarly, there are 255 y positions, with 192 of these visible on the screen beginning with 32 at the top. (These x and y values may vary slightly depending on the adjustment of your television receiver.) VBLANK PM assumes that you are referring to the upper left corner of your player whenever you POKE new x and y coordinates.

Lines 170 and 180 illustrate the use of the pseudo-random number function to determine the next set of x and y coordinates. Line 190 provides a small delay between player movements. Delete the FOR and NEXT statements if you want to see how fast – and easy – it is to move players.

Well, who said player/missile graphics had to be anything but fun?! Give VBLANK PM a try in one of your current programs to add a little zip; or try it in your next graphics project.

Oh, we almost forgot that we promised you a surprise regarding why you might want to change the height of a player. VBLANK PM has a few more features which allow you to animate the movements of your players – so turn to the next article!

Program.

```
90 REM * VBLANK PM DEMO *
100 POKE 752,1:? CHR$(125):SETCOLOR 2
    ,0,0
110 PCOL0=216:PCOL1=56:REM Color of p
    layers
120 GOSUB 1000:REM Initialize VBLANK
    routine
130 POKE PLL,8:POKE PLL+1,8:REM Heigh
    t of players
140 POKE PLX,108:POKE PLY,102:REM Pla
    yer 0's initial position
150 POKE PLX+1,108:POKE PLY+1,72:REM
    Player 1's initial position
160 REM Let players dance!
170 POKE PLX,RND(0)*159+48:POKE PLY,R
    ND(0)*191+32
180 POKE PLX+1,RND(0)*159+48:POKE PLY
    +1,RND(0)*191+32
190 FOR I=1 TO 75:NEXT I:GOTO 170
200 END
1000 REM * INITIALIZE VBLANK *
1010 FOR I=1536 TO 1706:READ A:POKE I
     ,A:NEXT I
1020 FOR I=1774 TO 1787:POKE I,0:NEXT
      I
1030 PM=PEEK(106)-16:PMBASE=256*PM
1040 FOR I=PMBASE+1023 TO PMBASE+2047
     :POKE I,0:NEXT I
1050 FOR I=PMBASE+1025 TO PMBASE+1032
     :READ A:POKE I,A:NEXT I
1060 FOR I=PMBASE+1281 TO PMBASE+1288
     :READ A:POKE I,A:NEXT I
1070 POKE 704,PCOL0:POKE 705,PCOL1:PO
     KE 706,PCOL2:POKE 707,PCOL3
1080 PLX=53248:PLY=1780:PLL=1784
1090 POKE 559,62:POKE 623,1:POKE 1788
     ,PM+4:POKE 53277,3:POKE 54279,PM
1100 X=USR(1696)
1110 RETURN
2000 REM * VBLANK INTERRUPT ROUTINE *
```

```
2010 DATA 162,3,189,244,6,240,89,56,2
     21,240,6,240,83,141,254,6,106,14
     1
2020 DATA 255,6,142,253,6,24,169,0,10
     9,253,6,24,109,252,6,133,204,133
2030 DATA 206,189,240,6,133,203,173,2
     54,6,133,205,189,248,6,170,232,4
     6,255
2040 DATA 6,144,16,168,177,203,145,20
     5,169,0,145,203,136,202,208,244,
     76,87
2050 DATA 6,160,0,177,203,145,205,169
     ,0,145,203,200,202,208,244,174,2
     53,6

2060 DATA 173,254,6,157,240,6,189,236
     ,6,240,48,133,203,24,138,141,253
     ,6
2070 DATA 109,235,6,133,204,24,173,25
     3,6,109,252,6,133,206,189,240,6,
     133
2080 DATA 205,189,248,6,170,160,0,177
     ,203,145,205,200,202,208,248,174
     ,253,6
2090 DATA 169,0,157,236,6,202,48,3,76
     ,2,6,76,98,228,0,0,104,169
2100 DATA 7,162,6,160,0,32,92,228,96
3000 REM * Draw players 0 & 1 *
3010 DATA 6,6,8,126,195,195,195,126
3020 DATA 126,195,195,126,24,126,126,
     24
```

Animation And P/M Graphics

Tom Sak and Sid Meier

This article builds upon the vertical blank routine in the preceding piece, showing how to animate players with pre-drawn shapes.

You're already familiar with the Atari's ability to rapidly move a player from one location to another. But there are many times when you would like to do more than simply move a player; you'd like to give it lifelike motion, or animation. Spend a few minutes and learn how you can achieve these effects with far less effort than you might have imagined.

The art of bringing life to still pictures is much older than many of us realize. The production of books which contained moving pictures was well established before the invention of the motion picture camera and projector. The effect of moving pictures was typically accomplished by rapidly flipping the pages of a booklet containing simple character drawings, making them seem to spring to life.

Walt Disney and numerous other animators have produced this illusion of motion by drawing series of pictures in which each picture differs from the previous one only in a very small detail, a subtle displacement of each moving element. The pictures are then photographed for subsequent projection.

For example, an animator, using a sequence of drawings, draws a man who appears to raise his arm away from his side. The first drawing would show the man facing you with both arms at his sides. The second picture differs only in that one arm is now slightly away from the man's side. The next picture shows the arm slightly further away, and so on through the sequence of drawings.

Animate With Only Four Drawings

As each picture in the series is viewed in rapid succession, by flipping through the stack of drawings, the figure appears to be

raising his arm away from his side. A motion picture film consists of an analogous sequence of pictures which also provide the illusion of motion when they are projected and viewed in rapid succession.

As you can well imagine, a very large number of drawings is required to produce even a relatively short motion picture sequence. Since you're not about to adapt *Fantasia* for the small screen attached to your Atari, we will show you a way to use only four drawings, repeated in a cyclical pattern, to produce the illusion of motion. This is a very effective shortcut which makes it practical to adapt the animator's techniques to your BASIC program.

Now for some Atari animation. There is no question that our artistic creativity and graphic talents may never rival those of Walt Disney, but we will endeavor to adapt the basic animation technique which he popularized in order to move four "cowboys" from right to left across your television screen, totally out of step with each other.

For illustrative purposes we'll begin by moving only one cowboy. Program 1 accomplishes this objective by using the automatic player/missile graphics manipulation of the vertical blank interrupt routine which we discussed in the preceding article. Those who have entered the example program in that article will be pleased to know it already contains the animation features described here.

Program 2 adds complexity to the one cowboy program, illustrating the asynchronous movement of four players. Developing an understanding of the more complex program won't be too difficult once you've grasped the concepts in Program 1.

Reviewing Vertical Blank Interrupts

An elementary understanding of our vertical blank interrupt routine, VBLANK PM, is a prerequisite. Here we will review highlights of our previous article.

VBLANK PM is a machine language subroutine which occupies a portion of memory page six. It is initialized by a single BASIC USR function call which causes VBLANK PM to notify the operating system of both its presence and its desire to be automatically invoked during each vertical blank interrupt.

Prior to initialization, a 2K (2048) byte memory allocation must be made for the storage of players, and the players must be drawn. Following initialization, a POKE of the x-axis (hori-

zontal) and y-axis (vertical) screen coordinates is all that is required to cause a player to be automatically moved during the next vertical blank period, or approximately every 60th of a second.

We hinted in the previous article that VBLANK PM has an animation feature just waiting to bring life to your players. All you need do is supply a few more drawings. The drawings and the current display image are contained in the 2K byte storage block.

Players Are Stored As Separate Images

Figure 1 depicts the memory allocated for the storage of players (see line 1030 in Program 1; memory allocation is explained in our earlier article). The current displayed image of player zero resides at locations PMBASE + 1024 through PMBASE + 1279; player one's homestead is PMBASE + 1280 through PMBASE + 1535, and so on for the other two players.

To achieve the animation, you need more than one image of each player, so the lower 1K (1024) locations (PMBASE through PMBASE + 1023) of the 2K byte storage block are used to hold the necessary set of drawings. Each player's drawings are stored in an area of memory beginning at a location which is 1K bytes below (lower memory address) the player's position in the upper 1K portion of the 2K byte storage block. A drawing is copied to the upper 1K portion by VBLANK PM when it is to be displayed. As a matter of fact, you won't draw anything at all in the upper 1K locations, but will let VBLANK PM look after this chore for you.

For example, all of the player zero drawings reside at the 256 locations beginning at PMBASE. The currently displayed image of player zero resides at locations PMBASE + 1024 through PMBASE + 1279. The drawings for player zero are stored 1024 locations below this point, which is equal to PMBASE + 1024 minus 1024, or simply PMBASE. The player one drawings begin at PMBASE + 256, or (PMBASE + 1280)-1024, and so on for players two and three at locations PMBASE + 512 and PMBASE + 768, respectively.

A note of caution: we mentioned in the previous article that you could use the lower 1K bytes for your own purposes without disturbing anything. This is true only when the VBLANK PM animation feature is not going to be used. We hope that you've not been led too far astray!

At the risk of stating the obvious, we'd like to mention that as soon as you've decided to use more than one drawing per player – which you must do in order to achieve the animation – you can no longer have a player which is 255 lines tall. This is true because there are only 256 locations in which to store all of the drawings necessary to animate a single player. The first position, location zero, of each storage bin is reserved for a reason discussed later.

Initialize The Vertical Blank Routine

Now let's turn our attention to Program 1. Line numbers ending in zero are the same as in the preceding article and, for those who previously keyed the lengthy DATA statements containing VBLANK PM, we've made no changes to the machine language subroutine.

Lines 105 through 205 are the main program which causes our ragtag cowboy to meander across the screen. The BASIC code required to load and initialize VBLANK PM is found on lines 1000 through 1110. The VBLANK PM machine language subroutine is represented as DATA in lines 2000 through 2100. Finally, lines 3005 through 3045 contain the four drawings, used to describe a single player.

Before reviewing the main program, we'll go over the initialization subroutine which performs three functions: load VBLANK PM, load the player's drawings, and initialize VBLANK PM.

Lines 1010 and 1020 cause VBLANK PM to be read from DATA statements and POKEd into memory page six. A more memory-efficient method of representing VBLANK PM is the use of a string variable instead of DATA statements. Using this alternative, you continue to POKE the VBLANK PM code into page six, but from the string variable instead of from DATA statements.

You would save memory because only a single byte of memory is required in the string variable assignment statement to represent a byte of machine language code. In the DATA statement, as many as three bytes may be required for the same thing. For certain other machine language code applications, you can directly execute from the string, eliminating the need to POKE the code into another memory location.

How The Animation Works

Line 1030 acquires the 2K byte memory storage block, and line

1040 assures that the upper 1K byte display portion is cleared. Lines 1045 through 1065 are responsible for reading and storing the player's drawings in the lower 1K byte portion of the storage block. The four drawings of a cowboy are illustrated in Figure 2; you see now why Disney Productions can rest easy!

Notice that in line 1045 the first location in which the first drawing is stored is established as one byte above PMBASE; you will learn why this is necessary in a minute. The FOR statement on line 1055 assures that four drawings (zero through three) are read and stored. Each drawing is 24 lines tall, so we begin the FOR loop on line 1065 with the base of the first drawing offset by 24 bytes for each previous drawing stored. Since each drawing consists of 24 bytes, the loop is completed by adding 23 to the starting point.

Line 1075 designates the player's color. Line 1080 establishes the locations to be POKEd to change the player's x-axis and y-axis screen coordinates (PLX and PLY) and to set the length (height) of the player (PLL).

The x-axis screen display positions for players zero, one, two, and three are indicated by POKEs to PLX, PLX+1, PLX+2 and PLX+3, respectively. The analogous situation is true for setting the player's y-axis coordinate (PLY, PLY+1, ...) and the player's height (PLL, PLL+1, ...), and for selecting the next drawing to be displayed (PDR, PDR+1, ...).

PDR is defined on line 1095 and is used to select the next drawing to be used as the player's current display image. VBLANK PM is responsible for copying the drawing to the appropriate location in the upper 1K byte portion of the 2K byte storage block. A value in the range of one to 255 is POKEd into PDR to indicate the bottom-most line of the selected drawing. The most recent value POKEd into PLL indicates the number of bytes (the height of the player) to be copied.

VBLANK PM Must Announce Itself

A value of zero POKEd into PDR signals VBLANK PM to continue to display the current image. This is why we were careful to avoid location zero when loading the first drawing. VBLANK PM sets PDR to zero automatically after it copies a drawing to the upper 1K byte display area.

Location 1771, POKEd in line 1095, is a location in VBLANK PM which must contain the memory page number of the first page in which drawings are stored. Location 1788, referenced

on line 1090, is also in VBLANK PM, and must contain the page number of the beginning of the upper 1K byte current display portion. (These parameters afford even greater flexibility to VBLANK PM, features which are beyond the scope of this discussion.)

The other POKEs on line 1090 are associated with the Atari's player/missile graphics mechanism which is described in numerous other articles.

VBLANK PM is initialized on line 1100. This is the only explicit BASIC function call to VBLANK PM which is required. As a result of this call, VBLANK PM will register its intention to become a part of the vertical blank interrupt process with the operating system.

Inside The Main Routine

Turning our attention to the main program, we start with line 105, which establishes the television screen background, or playfield. It is important that you always define a graphics mode (execute a graphics statement) before you initialize VBLANK PM; if you fail to follow this sage advice, you are likely to be plagued by a strange flashing vertical bar on your screen.

It doesn't matter which graphics mode is specified since Atari players are independent of the mode. Graphics mode one is chosen to provide a text window to serve as a walkway for our strolling cowboy. Line 125 sets the y-axis position of the cowboy so he appears to walk on top of the text window. The player's height is also established on line 125.

The animation is performed by lines 135 through 205. These lines should be relatively easy to comprehend once you have a mental picture of the way in which the drawings were stored during the initialization procedure. The variable DRAW, initialized as one on line 135, selects the next drawing to be used as the current display image.

Lines 145 and 165 control the right to left motion of the cowboy by using the index variable I as the x-axis coordinate of the player. The POKE to PDR on line 185 selects the next drawing to be displayed, and the calculation on line 195 results in the selection of the drawing to be used in the next cycle when the cowboy takes his next step.

The IF statement on line 195 assures that after the fourth drawing is used, the program will cycle and begin anew with the first drawing. The FOR loop on line 205 controls the speed

with which the cowboy strolls across the screen. A maximum value of 30 results in a movement which you might describe as a brisk walk. The larger the maximum value of this delay loop, the slower the pace of the player.

The cowboy will continue to walk across the screen until you stop the program. Incidentally, the program does not gracefully turn off the Atari's player/missile graphics mechanism, so you are well advised to press SYSTEM RESET to remove the undesirable residue from the screen. (POKE 53277,0 turns off the player/missile gracefully.) Be patient when the program is started, since it takes more than ten seconds for the initialization procedure.

Four Heads Are Better Than One

And that's almost all there is to animation! Are you ready to tackle a little bit more challenging project? Program 2 represents enhancements to the program we've been reviewing. It uses all four players and, while it causes them to walk out of step with each other, it employs only the same four drawings.

Program 2 modifies seven lines and adds two more. The changed lines are: 125, 165, 185, 205, 1045, 1055 and 1075; lines 155 and 175 are new.

Line 1045 now includes a FOR statement to cause the drawings to be READ and POKEd in the storage area associated with the additional three players. Note also that the calculation of DRWBAS is revised to reflect the additional players. DRWBAS contains the address of the first byte of the drawing storage area containing the first drawing for the current player. As the value of the variable, I, in the FOR loop is indexed from 0 to 3, DRWBAS will take the values 1, 257, 513 and 769. The first byte, location 0, of each storage area is skipped for the reason mentioned earlier.

A RESTORE statement is added to line 1055 which resets the DATA pointer to reread the same drawings for each player. The modification to line 1075 is simply the addition of player colors for the new players.

Looking at the main program, line 125 now establishes the y-axis and height for four players rather than one. Line 155 is added to cycle through the x-axis movement and picture selection for all players.

In line 165 we've added a calculation to the x-axis positioning POKE to maintain a separation between the cowboys which

is equal to slightly more than the width of a single player as measured from the left-most edge of one player to the left-most edge of the following player.

Still Only Four Drawings

Line 175 is added to assure that a different drawing is used as the current display image for each player. The variable DRAW continues to determine the drawing to be selected for player zero. Study the statement, and you will discover that each player will be depicted by the drawing following that used for the previous player. That is, if player zero is pictured by the first drawing, then player one is illustrated by the second, player two by the third, and, finally, player three is displayed as the fourth drawing. A circular assignment is used so that the fourth drawing is followed by the first.

The delay loop is omitted from line 205 because the additional calculations needed for the added players consume sufficient time to maintain a reasonable pace for all four cowboys. You might want to experiment with a delay loop to further slow the action; better yet, consider using GET to accept a keystroke instead of employing a delaying FOR loop. The GET will allow you to step the players across the screen in order to study the animation technique.

Don't you agree that animation makes a world of difference in the use of player/missile graphics? I was fascinated when my more talented partner, Sid, gave me a half-dozen lines of cryptic BASIC statements to turn into an animation tutorial. The first time I saw them execute, I was mesmerized. Go ahead, type either program into your Atari; you'll be addicted too.

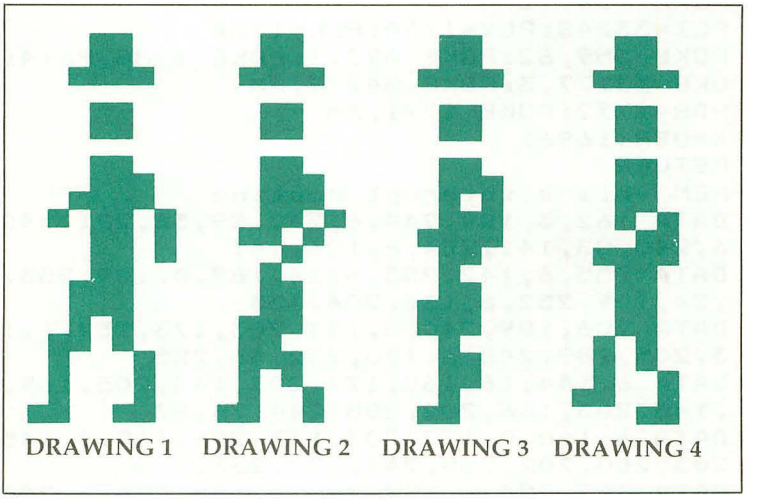
Figure 1.

8 bits wide

PMBASE	Unused (Player 0 - Drawing Storage)
PMBASE +256	Unused (Player 1 - Drawing Storage)
PMBASE +512	Unused (Player 2 - Drawing Storage)
PMBASE +768	Missiles (Player 3 - Drawing Storage)
PMBASE +1024	Player 0
PMBASE +1280	Player 1
PMBASE +1536	Player 2
PMBASE +1792	Player 3
PMBASE +2048	

( ) - VBLANK PM unique usage

Figure 2.



Program 1.

```
5 REM .... P R O G R A M{4 SPACES}O N E ....
105 GRAPHICS 1:SETCOLOR 2,1,8:SETCOLOR 4,8,4
    :POSITION 5,3:? #6;"animation":POSITION
    3,5:? #6;"demonstration"
120 GOSUB 1000:REM initialize vb routine
125 POKE PLY,169:POKE PLL,24
135 DRAW=1
145 FOR I=212 TO 10 STEP -1:REM move rt to 1
    ft horiz
165 POKE PLX,I:REM new position
185 POKE PDR,DRAW:REM new drawing
195 DRAW=DRAW+24:IF DRAW>73 THEN DRAW=1:REM
    select next drawing
205 FOR DELAY=1 TO 30:NEXT DELAY:NEXT I:GOTO
     145
1000 REM INITIALIZE VBLANK PM SUBR
1010 FOR I=1536 TO 1706:READ A:POKE I,A:NEXT
      I
1020 FOR I=1774 TO 1787:POKE I,0:NEXT I
1030 PM=PEEK(106)-16:PMBASE=256*PM
1040 FOR I=PMBASE+1023 TO PMBASE+2047:POKE I
     ,0:NEXT I
1045 DRWBAS=PMBASE+1
1055 FOR J=0 TO 3:REM four drawings
```

```
1065 FOR K=DRWBAS+J*24 TO DRWBAS+J*24+23:REA
     D X:POKE K,X:NEXT K:NEXT J
1075 POKE 704,12
1080 PLX=53248:PLY=1780:PLL=1784
1090 POKE 559,62:POKE 623,1:POKE 1788,PM+4:P
     OKE 53277,3:POKE 54279,PM
1095 PDR=1772:POKE 1771,PM
1100 X=USR(1696)
1110 RETURN
2000 REM vblank interupt routine
2010 DATA 162,3,189,244,6,240,89,56,221,240,
     6,240,83,141,254,6,106,141
2020 DATA 255,6,142,253,6,24,169,0,109,253,6
     ,24,109,252,6,133,204,133
2030 DATA 206,189,240,6,133,203,173,254,6,13
     3,205,189,248,6,170,232,46,255
2040 DATA 6,144,16,168,177,203,145,205,169,0
     ,145,203,136,202,208,244,76,87
2050 DATA 6,160,0,177,203,145,205,169,0,145,
     203,200,202,208,244,174,253,6
2060 DATA 173,254,6,157,240,6,189,236,6,240,
     48,133,203,24,138,141,253,6
2070 DATA 109,235,6,133,204,24,173,253,6,109
     ,252,6,133,206,189,240,6,133
2080 DATA 205,189,248,6,170,160,0,177,203,14
     5,205,200,202,208,248,174,253,6
2090 DATA 169,0,157,236,6,202,48,3,76,2,6,76
     ,98,228,0,0,104,169
2100 DATA 7,162,6,160,0,32,92,228,96
3005 REM drawings 0, 1, 2 and 3
3015 DATA 0,12,12,30,0,12,12,0,12,14,30,45,1
     3,13,12,28,28,20,52,34,34,34,102,0
3025 DATA 0,12,12,30,0,12,12,0,12,14,14,13,2
     6,4,8,12,12,28,24,28,20,18,50,0
3035 DATA 0,12,12,30,0,12,12,0,12,14,10,14,3
     0,12,8,12,28,28,8,12,12,8,24,0
3045 DATA 0,12,12,30,0,12,12,0,12,12,12,10,6
     ,30,12,12,12,12,20,20,18,50,6,0
```

Program 2.

This program uses the Vertical Blank Player/Missile routine, so add lines 2000-3045 of Program 1 when you type it in.

```
5 REM .... P R O G R A M{4 SPACES}T W O ....
105 GRAPHICS 1:SETCOLOR 2,1,8:SETCOLOR 4,8,4
    :POSITION 5,3:? #6;"animation":POSITION
    3,5:? #6;"demonstration"
120 GOSUB 1000:REM initialize vb routine
125 FOR J=0 TO 3:POKE PLY+J,169:POKE PLL+J,2
    4:NEXT J
135 DRAW=1
145 FOR I=212 TO 10 STEP -1:REM move rt to l
    ft horiz
155 FOR J=0 TO 3:REM four players
165 POKE PLX+J,I+J*10:REM new positioN, main
    tain separation
175 NXTDRW=DRAW+J*24:IF NXTDRW>73 THEN NXTDR
    W=NXTDRW-96:REM select different drawing
     for each player
185 POKE PDR+J,NXTDRW:NEXT J
195 DRAW=DRAW+24:IF DRAW>73 THEN DRAW=1:REM
    select next drawing
205 NEXT I:GOTO 145
1000 REM INITIALIZE VBLANK PM SUBR
1010 FOR I=1536 TO 1706:READ A:POKE I,A:NEXT
      I
1020 FOR I=1774 TO 1787:POKE I,0:NEXT I
1030 PM=PEEK(106)-16:PMBASE=256*PM
1040 FOR I=PMBASE+1023 TO PMBASE+2047:POKE I
     ,0:NEXT I
1045 FOR I=0 TO 3:DRWBAS=PMBASE+I*256+1:REM
     four players
1055 RESTORE 3015:FOR J=0 TO 3:REM four draw
     ings
1065 FOR K=DRWBAS+J*24 TO DRWBAS+J*24+23:REA
     D X:POKE K,X:NEXT K:NEXT J:NEXT I
1075 POKE 704,12:POKE 705,128:POKE 706,48:PO
     KE 707,192
1080 PLX=53248:PLY=1780:PLL=1784
1090 POKE 559,62:POKE 623,1:POKE 1788,PM+4:P
     OKE 53277,3:POKE 54279,PM
1095 PDR=1772:POKE 1771,PM
1100 X=USR(1696)
1110 RETURN
```

Extending Player/Missile Graphics

Eric Stoltman

Here's another way to animate player shapes with a machine language routine. It's also valuable for instantly changing the shape of a player to fit the direction it's traveling.

One of the best features of the Atari is player/missile graphics. This article and example program will explain how to create excellent animation, such as a walking figure or a rotating ship, with just one player.

One way to perform animation is to alternate players back and forth, but problems arise. What if the player is moved up? The other players also have to be moved up. This takes time. Another method is to alternately POKE data into the player, thus changing its shape. This can be done slowly in BASIC or quickly and easily in machine language. This program will compare both the BASIC and machine language methods for changing the data of a player.

After a player is set up, additional data for other shapes must be stored in RAM. I prefer to use memory locations 256 to 511, since they are empty and are protected. This data can be manipulated by setting up pointers in an array. A subroutine can then easily retrieve this data and place it in the player's data area. This can be done in BASIC:

```
C=0: FOR A=PMBASE+512+Y TO PMBASE+519+Y:POKE
     A, PEEK(POINTER(FACING)+C) :C=C+1: NEXT A
POINTER(FACING)=Array containing addresses of data.
EXAMPLE: POINTER(1)=260,POINTER(2)=268, etc.
```

Or in machine language:

```
A=USR(XXX,PMBASE+512+Y,POINTER(FACING))
```

XXX = Address of Machine Language subroutine.

The machine language method is not only easier, but also executes 11 times faster and provides smoother motion.

The machine language code is relocatable and can easily be modified by changing the 22nd DATA element so more or less data can be POKEd into the player's data area.

In addition to providing animation, this subroutine can move a player up or down when the vertical value changes greatly. To do this, point to an empty area of RAM (thus erasing the player), and then change the vertical value and point to the desired data. An example would be if a player went off the top of the screen and, using the method mentioned above, quickly reappeared at the bottom.

I should point out that many false players, that is, data for alternate shapes, may be stored and rotated among the four players to provide excellent animation.

Line Numbers	Explanation
110-130	POKE machine language subroutine for changing player into player.
140-170	POKE data for additional shapes into memory.
180-190	Set up pointers to data.
200-250	Set up player.
270-330	If trigger is pressed, change player by machine language.
340-400	If trigger is not pressed, change player by BASIC.

Program.

```
20 REM THE "I" IN THE VARIABLE "POINT
   ER" IN LINES 190,250,320,390 SHOUL
   D BE A "1" AS "POINT" IS A RESERVE
   D WORD
100 REM ** INITIALIZATION **
110 FOR A=1536 TO 1560:READ I:POKE A,
    I:NEXT A:REM POKE DATA FOR MACHIN
    E LANGUAGE SUBROUTINE INTO MEMORY
     PAGE 6
120 REM ** MACHINE LANGUAGE DATA **
130 DATA 104,104,133,204,104,133,203,
    104,133,207,104,133,206,160,0,177
    ,206,145,203,200,192,8,208,247,96
140 REM ** ADDRESS OF PLAYER DATA **
150 FOR A=260 TO 323:READ I:POKE A,I:
    NEXT A:REM POKE DATA INTO PROTECT
    ED RAM
155 REM ** PLAYER SHAPE DATA **
160 DATA 28,62,62,28,73,127,73,65,7,2
    3,39,88,154,36,8,16,240,38,47,127
    ,47,38,240,0,16,8,36,154,88,39,23
    ,7
170 DATA 65,73,127,73,28,62,62,28,8,1
    6,36,89,26,228,232,224,15,100,244
    ,254,244,100,15,0,224,232,228,26,
    89,36,16,8
180 REM ** POINTERS TO DATA **
190 DIM POINTER(8):FOR A=1 TO 8:READ
    I:POINTER(A)=I:NEXT A:DATA 260,26
    8,276,284,292,300,308,316
200 REM ** SET UP PLAYER **
210 GRAPHICS 0:POKE 752,1:POKE 710,0:
    POKE 559,46
220 A=PEEK(106)-8:POKE 54279,A:POKE 5
    3277,3:PMBASE=256*A:POKE 53256,1:
    X=124:Y=48
230 FOR A=PMBASE+512 TO PMBASE+640:PO
    KE A,0:NEXT A
240 POKE 53248,124:POKE 704,12:FACING
    =1
```

```
250 C=0:FOR A=PMBASE+512+Y TO PMBASE+
    519+Y:POKE A,PEEK(POINTER(FACING)
    +C):C=C+1:NEXT A
270 REM ** MACHINE LANGUAGE CHANGE **
275 IF STRIG(0)=1 THEN 340
280 POSITION 5,5:? "MACHINE LANGUAGE"
    ;
285 J=STICK(0)
290 IF J=15 THEN 270
300 IF J=11 THEN FACING=FACING-1:IF F
    ACING<1 THEN FACING=8
310 IF J=7 THEN FACING=FACING+1:IF FA
    CING>8 THEN FACING=1

320 D=USR(1536,PMBASE+512+Y,POINTER(F
    ACING))
330 GOTO 270
340 REM ** BASIC CHANGE **
345 IF STRIG(0)=0 THEN 270
350 POSITION 5,5:? "BASIC{11 SPACES}";
355 J=STICK(0)
360 IF J=15 THEN 270
370 IF J=11 THEN FACING=FACING-1:IF F
    ACING<1 THEN FACING=8
380 IF J=7 THEN FACING=FACING+1:IF FA
    CING>8 THEN FACING=1
390 C=0:FOR A=PMBASE+512+Y TO PMBASE+
    519+Y:POKE A,PEEK(POINTER(FACING)
    +C):C=C+1:NEXT A
400 GOTO 340
```

The Collision Registers

Matt Giwer

Collision registers allow you to detect overlaps between players, missiles, and other objects on the screen – an especially valuable feature for game programming.

When using player/missile graphics, the Atari Operating System sets aside 16 registers (memory locations) for determining collisions. These collision registers can be read to see if there has been a collision, and what the player or missile has collided with. This allows control of events or actions within the program – such as triggering explosions, for example. For complex games, we need detailed knowledge about the numbers in these registers.

Exactly what is a collision? In the player/missile graphics sense, a collision occurs when the CTIA or GTIA video display chip is directed to overwrite an ordinary screen graphic or a player/missile with another player/missile. As part of this overwriting process, the computer writes different numbers into its collision registers, depending upon what kind of overwrite has occurred. In other words, when two things are to be on the screen at the same place at the same time, a number will be written to these registers.

An important fact about a collision is that there must be an overwrite by at least one pixel, rather than as a ball colliding with a wall where touching is enough to be called a collision.

Also, the numbers in these registers will not return to zero during the vertical blanking period (the split-second a TV screen is blank between frames). The only way to return the collision registers to zero is to POKE a number into register 53278. This is called the HITCLR, for Hit Clear, register.

Refer to Table 1. This shows what happens when Players 0 through 3 collide with other players. The left-hand column shows the register number associated with the collision of the

player. The Player 0 through Player 3 across the top are the ones being collided with. The values in the table are those returned after the collision. For example, if Player 0 collided with Player 1, then PRINT PEEK(53260) would return the number 2. Also, PRINT PEEK(53261) would return the number 1, because the players have collided with each other and both registers will have a number written in them.

Look again at Table 1. Since Player 0 cannot collide with itself, the value remains zero. When you POKE a number into the HITCLR register, this collision register returns zero and remains zero until there is a collision with a player other than itself. This permits a register test such as IF PEEK(53260)>0 THEN [SOMETHING] rather than using three test statements for the values 2, 4, and 8. Thus, if you do not need to know exactly which player has been in a collision, you need not test for it.

Another aspect of this register is that it returns the sum of the collision values. That is, if Player 0 collides with more than one other player before you POKE HITCLR, then the number in register 53260 will be the sum of the numbers. If Player 0 collides with Player 1 and Player 2, then the collision register will contain 6; with Player 1 and Player 3, then the value will be 10; with Players 1, 2, and 3, the value will be 14. Note that in this case, as in Tables 3 and 4, these numbers are generated by setting bits 0 through 3 in the registers. Thus, in a machine language routine, the state of these registers can be determined by using a logical AND with a compare (CMP).

Table 2 shows the register values resulting from collisions of players with playfields drawn with the BASIC instruction COLOR. Depending upon the graphics mode, these can be character sets (normal or redefined), or PLOT and DRAWTO figures. There is no requirement that these actually be different colors since the real colors are controlled by registers 709, 710, and 711 and can be set to the same value. The playfield graphic needs only to be drawn by the COLOR instruction. The graphic might even be the same color as the background, and therefore invisible. Because these collisions return different numbers, it is possible to have different responses depending upon the graphic collided with. Thus a room may have walls drawn with COLOR 1 and a door with COLOR 2. When the collision register value returned is 1, the player will not be able to pass through the wall, but when the value is 2, the player can pass through.

Tables 3 and 4 are similarly structured and complete the register information. All have the same general characteristics. The number is generated by the setting of bits 0 through 3 in the registers, so the returned decimal value is 1, 2, 4, or 8 if only one collision has occurred, or the sum of these numbers if multiple collisions have occurred. Note that since no two numbers add up to any third collision value, multiple collisions cannot be confused with single collisions. Even with multiple collisions there is always a unique number returned. These values will remain in the registers until HITCLR is POKEd.

Note also that several collision registers are not provided. Collisions between missiles and other missiles are not detected, nor are collisions among playfields. Nor, as in the case of player collisions, are there reciprocal registers for the other possible collisions. For example, there are registers to detect when missiles collide with players, but there are no reciprocal registers to determine if players have been hit by missiles. To some extent this dictates the character of the program. When a missile is fired, the *missile* registers must be tested.

Another characteristic of these registers is that even though a collision can be described as an overwrite, the registers fill regardless of the priority between players and playfields that has been selected by a POKE into register 623, the shadow (*duplicate*) of 53257. Setting these priority registers allows players to pass in front of or behind playfields, to simulate three-dimensional movement. But despite the priority selected, the register will respond as though there were no priority. Missiles will respond the same as their associated players. Thus when two objects are to be on the screen at the same place at the same time, the registers will change whether or not you can see both of the objects.

The collision registers also fill with the same number from each collision. Thus, if Player 0 collides with three separate objects all drawn with COLOR 1, register 53252 will still contain the value 1. No matter how many times the player collides with the different objects, the value will remain 1, until it collides with a playfield figure drawn with COLOR 2 or 3 or until HITCLR has been POKEd. Although different COLOR collisions will result in the sum, multiple collisions with the same COLOR do not yield a sum.

This way you can draw several identical graphic figures with the same COLOR, and then PEEK the collision registers

for player collisions. When a collision is encountered, the location of the player can be used to determine where, among the many identical graphics, the next action (such as an explosion) is to occur.

Table 1. These values result from collisions among players.

	Play0	Play1	Play2	Play3
Player 0/Register 53260	-0-	2	4	8
Player 1/Register 53261	1	-0-	4	8
Player 2/Register 53262	1	2	-0-	8
Player 3/Register 53263	1	2	4	-0-

Table 2. These values result from collisions between players and playfield graphics.

	COLOR 1	COLOR 2	COLOR 3
Player 0/Register 53252	1	2	4
Player 1/Register 53253	1	2	4
Player 2/Register 53254	1	2	4
Player 3/Register 53255	1	2	4

Table 3. These values result from collisions between missiles and playfield graphics.

	COLOR 1	COLOR 2	COLOR 3
Missile 0/Register 53248	1	2	4
Missile 1/Register 53249	1	2	4
Missile 2/Register 53250	1	2	4
Missile 3/Register 53251	1	2	4

Table 4. These values result from collisions between missiles and players.

	Play0	Play1	Play2	Play3
Missile 0/Register 53256	1	2	4	8
Missile 1/Register 53257	1	2	4	8
Missile 2/Register 53258	1	2	4	8
Missile 3/Register 53259	1	2	4	8

The Priority Registers

Bill Wilkinson

Those of you who have studied the *Atari Hardware Manual* have probably been overwhelmed by the number of "registers." There are registers that define the start of the character set, the origin of the display list, the graphics mode, the Direct Memory Access modes, the amount of horizontal and vertical scrolling, the colors of the playfields and players, and much, much more. And yet, for the most part, the use of these registers is fairly clear and distinct, one from another. For example, you certainly know that a register called "COLPF2" (COLor of PlayField 2) wouldn't be used to define the start of the player/missile graphics area (that's the job of "PMBASE").

But are there some exceptions to this "separation of work" philosophy? Well, one might seem to be "DMACTL", which can turn players and missiles on and off, turn the playfield on and off, and determine whether players and missiles have single- or double-line resolution. And yet the register's varied functions can all be justified under the heading of "Direct Memory Access ConTroL".

However, there is one register which does seem to have several unrelated functions. From the viewpoint of both software and article authors, the functions which this register controls are all extremely interesting. Yet little has been written about this register's dominant function. Naturally, I hope to change that.

What's In A Name?

The register in question was named "PRIOR" by Atari. It is important enough that it was even given an Operating System shadow location, "GPRIOR", at 623 (decimal). (A "shadow location" is a RAM location into which a program may store a value actually intended for a hardware register. During the vertical blank interrupt processing – that is, every 60th of a second with U.S. NTSC television and every 50th of a second with the European PAL system – the Operating System ROM code moves each shadow location to the corresponding register.

The purpose of this is usually to force the register load to occur at a time when the screen is blanked, thus avoiding strange and bizarre visual effects.)

So just exactly what does this location control? Several things. Let's start with a recap of the table in Atari's *Hardware Manual*.

Bits 0-3	Priority Select
Bit 4	Fifth Player Enable
Bit 5	Multiple Color Player Enable
Bits 6-7	Special GTIA Display Mode Selects

Well, at least it's true that half of the register is used for "PRIORity" work, but wow! Look at all the other goodies.

This article is not going to deal with the extra GTIA modes (GRAPHICS 9, 10, and 11 in BASIC parlance), but I would like to invite you to court disaster. If your machine has a GTIA chip (as do most recent models), then sometime when you have a nice listing or display on the screen, try using POKE 623,128 or POKE 623,64 or POKE 623,192. If you do it when a listing is on the screen, you will quickly see why Atari doesn't provide mixed mode (text window) graphics for modes 9, 10, and 11. (Incidentally, if nothing happens when you do the POKEs, you don't have a GTIA.) Oh, yes, you can POKE 623,0 to return to normal.

The real "discoveries" to be explored in this section are the influences of bits 4 and 5, but before I delve into them, I would like to at least touch on the capabilities of the priority select bits.

Who's On First? What's On Second?

Atari states that each of the four priority select bits will produce a single type of supposedly mutually exclusive priority ordering of the various players and missiles. For example, if bit 0 is turned on (POKE 623,1), all the players have "priority" over all the playfields (and everything always has priority over the background). Within the group of players, a lower-numbered player has priority over a higher-numbered one.

But what does it mean to say something has priority? Simply this: since each player moves independently not only of the playfield but also of every other player, it is possible for one or more objects to appear at the same spot on the display screen at the same time. "Priority" simply answers the question of who gets displayed first, second, etc. An important point to re-

member: only those parts of player stripes containing "on" bits (i.e., only those parts to be made visible) participate in the priority contest! The "off" portion of all players is totally ignored by the system.

In theory, a lower-numbered playfield takes precedence over a higher-numbered one. In actuality, there is no way for two playfields to occupy the same video space, so the distinction is a moot one (except for one obscure case, as we shall see below). Anyway, here is a table of the various available priorities:

Bit 0 (POKE 623,1): All players, all playfields.
Bit 1 (POKE 623,2): Players 0 and 1, all playfields, players 2 and 3.
Bit 2 (POKE 623,4): All playfields, all players.
Bit 3 (POKE 623,8): Playfields 0 and 1, all players, playfields 2 and 3.

Although the bits are described as mutually exclusive, there is nothing to prevent you from turning on two or more (e.g., POKE 623,7). The theory says that if you overlap any two objects whose priorities are "in conflict" as a result of the multiple bits being on, the display will turn black in the overlap region. This would mean that if you used "POKE 623,5" all players would turn black except when over the background. I have yet to see a program use this to advantage, but no doubt somebody will someday.

And one final note: the whole reason for having priorities is so that you can gracefully control what happens to players when they meet the background display. Imagine that we select bit 3. Then if we have an airplane made from players, clouds in playfield color 0 or 1, and mountains in playfield color 2 or 3, the plane will fly in front of the mountains and yet behind the clouds!

When Is A Missile Not A Missile?

Atari would have you believe that the answer is "When it's a player." The name of Bit 4 of PRIOR is enticing: "Fifth Player Enable." Wow! We can turn this bit on and all of a sudden we have five players, right?

By default (i.e., if this bit is not set), each player takes on the same color as its "parent" player. Turn this bit on, and all missiles share a single common color. And that is all this bit does. Period.

What color do the missiles use? Playfield Color 3. Why was that color chosen? Because, in the normal graphics modes,

the only way to get that color is in GRAPHICS 1 or GRAPHICS 2, the large text modes. Notice that there are five SETCOLORs available, even though most graphics modes allow only two or four colors to be displayed. Since the background (SETCOLOR 4 in four-color modes) is always one of these (COLOR 0), one of the SETCOLORs – in point of fact, SETCOLOR 3 – goes unused. Until now.

So how can you use the four missiles as a single player? From BASIC, it's not easy. Each missile still retains its own independent horizontal position. Each missile still retains its own independent horizontal width. And, naturally, you still have to move the player/missile stripe vertically (although that is easier to do for a player than for a missile). This means that, for such a simple operation as moving the new "player" horizontally to (say) position X, you must perform this series of BASIC statements:

```
POKE HPOSM0,X
POKE HPOSM1,X+2
POKE HPOSM2,X+4
POKE HPOSM3,X+6
```

And even *that* only works if you have previously specified that all the missiles have single horizontal width (admittedly the SYSTEM RESET default).

Any other caveats? Yes. The missiles still act independently as far as the collision registers are concerned. And the Atari documentation claims that the priority of the new "player" is the same as that of Playfield 3, but that's only partially true. In particular, when considering which *player* has priority, it is true that the fifth player behaves according to the chart. But, strangely enough, the fifth player *always* has priority over *any* playfield color. This might imply some interesting consequences for a creative game designer.

So, is this "Fifth Player Enable" bit actually useful? From pure BASIC, I think not. With some machine language support, probably. And, of course, from pure machine language (or C or FORTH or Pascal), you can probably do some really interesting things. After all, how many moving objects do you see in *Pac-Man* or *Jawbreaker*?

All Colors Of The Rainbow

How many times have you read that, even though the Atari can display 128 different colors, it can display only four of those at a time? Well, you probably know by now that it's simply not true. First of all, if you have a GTIA, it's actually possible to display 16 different colors on the playfield in two of the GTIA graphics modes, and nine colors in the other mode. But let's go after the maximum number of colors possible.

To begin, choose GRAPHICS 11. That will get you 16 possible hues, each with the same luminance. Great.

Now, let's add four players, each with its own color and each different from any of the playfield colors. Now we are up to 20 colors!

But we're not done: why not use the "Fifth Player Enable" bit to get yet another color (since Playfield 3 is not involved in the GRAPHICS 11 playfield display). Twenty-one colors!

But.... Yep, you guessed it, there's more. Remember that bit titled "Multiple Color Player Enable"? If you turn on that bit (POKE 623,32 or POKE 623,49 to also do all we just described), then pairs of players that overlap will generate yet another color in the overlap region! Actually, the valid overlap pairs are limited to Player 0 joined with Player 1 and/or Player 2 overlapping Player 3. (If Player 0 or 1 overlaps Player 2 or 3, the priority rules still apply.)

So there you have it. Twenty-three colors displayable at the same time. And I believe that, without resorting to display list interrupts or similar chicanery, that is the Atari computer's maximum.

But that isn't the real reason I introduced the Multiple Color Player Enable. Do you have an object that you want to animate that is just begging to be multicolored? Here is your method of implementation. And if you need even more than three colors, you could use all four (or three, or five) players to display it. If you need to animate only a single object, the multicolor players could give you some very nice control over both detail and color.

Unfortunately, to do effective multicolored work, you must devote two players to each animated object. Even if you use the "fifth player" trick, this limits you to three moving objects. Yet I can't help but think that this is a solution waiting for a problem. Come on, game designers, get to work.

We need to make a couple of points before we leave this subject: by default, missiles behave the same as their parent players. That is, if the multiple color players are enabled and missile 0 overlaps missile 1, then the overlapped region will result in the third color being displayed. But Atari did it right: if you choose the fifth player option, the missiles all take on that fifth player color and the missile overlaps have no effect.

And, finally, I would like to say that I have not yet figured out how to predict what the third color, displayed in the overlap region, will be. The hue and intensity in the region do not seem to directly relate to that of the overlapping colors. Generally, the intensity is brighter than the duller of the two, but it is not always brighter than the lighter. It seems to me that the resulting color might be some sort of vector addition of the other two colors, but I don't know that. Want to write an article for **COMPUTE!** ? Here's a topic for a good one.

A Hair-Trigger Reaction

Before we leave this section, I think an example of some of the capabilities we have been discussing might be in order. In the discussion that follows, we will be referring to the diagrams in the figure.

The aim here is to design a "crosshair" using players. Obviously, we could make an adequate crosshair with a single player, but that wouldn't give us a chance to try out what we've been discussing. So examine the first three parts of the figure.

First, we define Player 0. Note that I have supplied the hexadecimal and decimal values necessary to produce each line (i.e., byte) of the player for those of you who would like to actually try this. Anyway, notice the gaps in what is otherwise a nice cross shape.

But if we now define Player 1 as shown, and purposely overlap the two players correctly, notice that we get a nice, two-colored crosshair. This is version one.

For version two, we assume that PRIOR = 1 (POKE 623,1). We redefine Player 1 so that its "hole" in the middle is gone. And yet, when we overlap it with the original Player 0, we get the same results we did with version one. Why? Simply because Player 0 has higher priority than Player 1, so the bits in the middle of Player 0 effectively override those in Player 1.

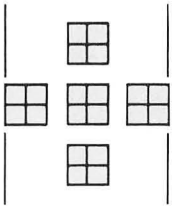
But finally we get to version three. The only difference between this version and version two is that now we have

turned on the Multiple Color Player Enable bit. Lo and behold, since the very center has bits turned on in both Player 0 and Player 1, we achieve yet a third color. A most satisfactory and colorful crosshair.

So there you have it. Some fairly impressive displays have been generated using players, and yet I have still not seen one that takes advantage of all the features we have discussed here. So, if you are a "do-er," do it!

Overlapping Players for creative results.

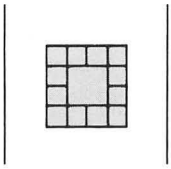
Player 0



Hex	Decimal
18	24
18	24
00	0
DB	219
DB	219
00	0
18	24
18	24

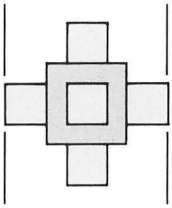
Player 1

Version 1



Hex	Decimal
3C	60
24	36
24	36
3C	60

The combined result

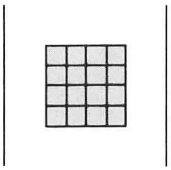


A "crosshair"

version one

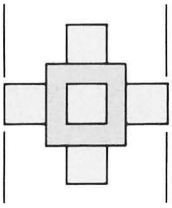
Player 1

version 2



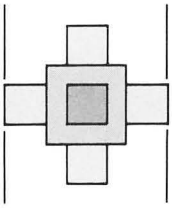
Hex	Decimal
3C	60
3C	60
3C	60
3C	60

The combined result



A "crosshair"

version two
(see text)



A "crosshair"

version three
(see text)

Chapter 6

Advanced Graphics Techniques

GRAPHICS 8 In Four Colors Using Artifacts

David Diamond

A painless, no-POKE method for mastering Atari high resolution, four-color graphics from BASIC.

Contrary to what the *Atari BASIC Reference Manual* states, GRAPHICS 8 is a true four-color mode (five colors if you count the border). Other articles have shown you how to obtain 16 or 128 colors by PEEKing, POKEing, and using machine language subroutines to fake out the operating system. This article is different. You can paint with four colors using simple, straightforward BASIC programming.

You probably have noticed that patterns drawn in GRAPHICS 8 often contain spurious colors. Atari sketches your television's resolution to its limits and the extra hues do sneak in.

The spurious colors seem random because they are appearing within a random pattern. They are, however, well-behaved. They can be harnessed, controlled, and used for brilliant displays.

Before I get into the details, try the following demonstration program:

```
10 GRAPHICS 8:COLOR 1
15 R=50
20 FOR X=-R TO R STEP 2
30 Y=SQR(R*R-X*X):REM Formula for a c
   ircle
40 PLOT 100+X,100+Y:DRAWTO 100+X,100-
   Y:REM Circle #1
50 PLOT 151+X,100+Y:DRAWTO 151+X,100-
   Y:REM Circle #2
60 NEXT X:FOR I=1 TO 350:NEXT I
70 FOR C=0 TO 15
```

```
80 SETCOLOR 2,C,4:SETCOLOR 4,15-C,8
85 FOR I=1 TO 350:NEXT I
90 NEXT C
```

Surprise! You have five vivid, solid colors on the screen at the same time. Now let's take a look at that program:

Line 10 – Straight GRAPHICS 8. Standard color defaults.

Line 15 – "R" is the radius of a circle.

Line 30 – This is the formula for a circle: $X^2+Y^2=R^2$. ("R*R" is a little faster than "R∧2").

Line 40 – This draws the first circle. It is vertically cross-hatched to fill it in with a solid color.

Line 50 – This draws the second circle. *But why is it a different color* from the first circle?

Line 20 – Ah, here begins the secret: "STEP 2". Before reading further, change it to "STEP 1" and rerun the program.

Lines 40,50 – Here is the second half of the secret: "100+X" is an *even* offset. "151+X" is an *odd* offset. Change both occurrences of "151" to "150" on line 50, and see what happens. (Remember to set line 20 back to "STEP 2".)

Lines 70-90 – These lines show you the wide range of color combinations available. Of course, when you are varying the luminance level, there will be even more.

Alternating Colored Fields

Without any additional programming lines, the circles can easily be changed into beach balls with alternating bands of color. Make sure that line 20 says "STEP 2", and change lines 40 and 50 as follows:

```
40 PLOT 98+X,100+Y:DRAWTO 101+X,100-Y
   :REM Circle #1
50 PLOT 147+X,100+Y:DRAWTO 150+X,100-
   Y:REM Circle #2
```

Changing the slope of the cross-hatching by a single horizontal point will add or remove one band of color. Increment the DRAWTOs by one horizontal point, and see what happens:

```
40 PLOT 98+X,100+Y:DRAWTO 102+X,100-Y
   :REM Circle #1
50 PLOT 147+X,100+Y:DRAWTO 151+X,100-
   Y:REM Circle #2
```

Although the quirk that provides us with the extra colors seems somewhat magical, the reason for the varied solid colors is not. Remember that the "colored-in" areas are really comprised of finely separated, vertical lines. To better see what is happening, spread those lines out into a large grid for easier inspection:

```
10 GRAPHICS 8:COLOR 1
20 FOR X=10 TO 160 STEP 15
30 PLOT X,1:DRAWTO X,160
40 PLOT 1,X:DRAWTO 160,X
50 NEXT X
```

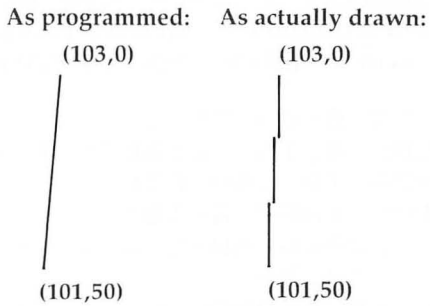
This isolates your three colors. The even column vertical lines are one color. Odd column vertical lines are a second color. Horizontal lines are the third color. (The background is the fourth, and the border is the fifth.)

Line 20 controls the colors. Try "FOR X=10 TO 160 STEP 14" and try "FOR X=9 TO 160 STEP 14".

When two adjacent lines touch each other ("FOR X=...STEP 1"), the two colors blend into the official color for graphics mode 8. Another way to look at it is that there are no longer separate lines when they touch, but rather a solid field of pixels.

The Alternating Color Phenomenon

The beach ball display, with its alternating bands of color, takes advantage of the fact that, with a pixel matrix, one cannot draw nearly vertical pure diagonal lines. Instead a series of shorter vertical lines are drawn, as shown below:



You can see that the three vertical line segments are drawn on odd, even, and odd columns, respectively, thus alternating colors.

Why Multiple Colors

The horizontal resolution limit of a television set is about 160 unique points. This is because on any one line of the television tube surface there are 160 sets of phosphor points which emit light when struck by the scanning electron beam. Each set actually contains three separate phosphor points – one that glows blue when struck, one that glows green, and one that glows red. Combinations of these dots in various intensities create the myriad of colors available.

Atari, in order to provide finer resolutions than 160 bytes across, plots 320 points across the screen – two for each set of color dots. (This is referred to as a *half color cycle*, or a *half color clock*.) Thus, even-column points will turn on the left portion of the three color phosphors, and odd-column points will turn on the right portion, producing alternating colors. The effect is referred to as *artifacting*.

Diagonal Lines

Diagonal lines, ranging from vertical to almost 45 degrees, contain vertical components, and are therefore subject to the artifacting effects described above. However, when these lines are drawn on top of a "...STEP 2" solid colored field (such as demonstrated in the above programs), much of the spurious color effect is minimized, so that the "official" color for graphics mode 8 will be seen. If the background is dark, a medium intensity line will appear light (whitish). If the background is bright, a medium intensity line will appear dark (often a rich chocolate brown).

The bold splashes of multiple solid colored shapes can thus be combined with the more delicate effects of intersecting diagonal lines, as in the following demonstration program:

```
10 GRAPHICS 8:COLOR 1
20 SETCOLOR 4,15,10:SETCOLOR 2,0,15
30 FOR A=20 TO 140 STEP 2
40 IF A=100 THEN A=101
50 PLOT 65,20:DRAWTO A,1:DRAWTO A,A:D
   RAWTO A+30,70
60 DRAWTO 65,A:DRAWTO 30,A+40:DRAWTO
   65,20
70 NEXT A
80 FOR I=1 TO 350:NEXT I
```

```
90 FOR COLOR=0 TO 15
100 SETCOLOR 2,COLOR,5:SETCOLOR 4,15-
    COLOR,10
110 FOR I=1 TO 350:NEXT I
120 NEXT COLOR
```

Moiré Patterns

No discussion of multiple colors would be complete without mentioning color moiré patterns. There are two types of moiré patterns. One type is the secondary pattern produced by the intersection of diagonal lines, such as is illustrated by demonstration Program 2, above. This type is not dependent on color for its effect. The second type is the subtle and delicate designs produced by shifts in color along diagonal lines. This type is dependent on the artifacting effect and is illustrated in the following program:

```
10 GRAPHICS 8:COLOR 1
20 FOR A=0 TO 319 STEP 3
30 PLOT 0,159:DRAWTO A,0
40 PLOT 319,0:DRAWTO 319-A,159
50 NEXT A
```

Notice that the pattern is whitest in the center, where the lines are not as steeply sloped, and also toward the upper right and lower left corners, where the lines are closest together. In addition to the white and the two artifacted colors, you may notice a fourth and fifth color along the top and bottom sections of the pattern. These extra colors are formed by a *visual* blending of the two artifacted colors. It is caused by the fact that the alternating colored areas are so close together that the eye has difficulty resolving them (a trick used by the Impressionists).

You can combine the various effects discussed in this article. Experiment with different color and intensity combinations. Blend in some dynamic color changes. You have a palette that any artist would envy.

Part I: Atari Video Graphics And The New GTIA

Craig Chamberlain

In this, the first of a three-part series on the inner workings of Atari graphics, the author reviews the computer's system of screen management and defines several important terms including color clock, playfield, mode line, *and* display list. *The next article includes techniques for using* color indirection, *a powerful graphics tool, and explores the new GTIA chip in detail. The final article demonstrates the capabilities of the GTIA. This new chip costs nothing if your Atari is still under warranty. If you have an older machine, the nearest authorized service center can install it for you.*

The GTIA is an exciting new graphics chip now being shipped in Atari 400/800 computers. Among its special features are a 16-color mode with a resolution eight times better than the Apple's, and the capability of generating 256 color variations. The GTIA chip provides three new graphics modes in addition to the normal 14, totally different, full-screen modes. This article defines a few terms relating to graphics, explains the normal graphics modes, then introduces the new modes provided by the GTIA.

ANTIC Is A Busy Chip

We all know that the Atari 400 and 800 have superior graphics capabilities. This has been achieved by designing special chips to handle video display tasks, taking that burden off the main microprocessor. In Atari computers these special chips are known as ANTIC and CTIA.

The ANTIC chip is actually an advanced DMA (direct memory access) controller that qualifies as a true microprocessor. It has an instruction set (mode lines and "load memory scan" operation), a program (the good 'ole display list), and data (dis-

play memory and character sets).

This special chip is a rather busy fellow. Its responsibilities include doing DMA for the display list, the display data (playfields), the character set, and player/missile memory. Besides that, it sets the playfield width, controls horizontal and vertical fine scrolling, keeps track of the vertical position of the scan beam, and handles non-maskable interrupts. It also supports a light pen.

The GTIA: Three New Modes

The other chip is the CTIA, or Computer Television Interface Adapted integrated circuit. This is the chip which handles all color and luminance (brightness) information to send to the television screen. This is a complicated process, but the chip designers at Atari got carried away and created whole new functions which we know as the player/missile graphics system. It is the CTIA which processes the horizontal position, size, priority, and color of the players. The CTIA also watches for player/playfield collisions, joystick triggers, and console keys. Like the ANTIC, it is a busy chip.

The new GTIA chip replaces the CTIA. Rumor has it that the "G" stands for George. Apparently some fellow named George was still not satisfied with all the special functions of the CTIA, and gave it the ability to generate three totally new graphics modes. When you find out what these new modes can do, I think you will appreciate "George" and his GTIA.

The three new modes are 9, 10, and 11. The operating system and, therefore, Atari BASIC, supports these new modes. But before describing all the features of these new modes, I want to define a few essential terms and review the normal graphics modes 0 through 8.

In order to fully understand Atari graphics, one must have a solid concept of how a television display is generated. And no discussion on "television theory" would be complete without a definition of the "color clock." The term *color clock* derives from the fact that there is a problem in measuring distances on a television screen. Different television sets have different screen sizes, with 9", 13" and 19" being common diagonal measurements. All television sets, however, have a scanning beam which translates a signal from the computer (or a TV station) into a picture on the screen.

The signal coming from the computer contains two characteristics. It has a frequency, which defines a color, and it has an amplitude, which defines the *luminance* of that color, often referred to as the brightness or intensity. These qualities of the computer signal affect the way in which the scanning beam shoots electrons at the phosphors on a television screen. This electron shooting process is done horizontally, one line at a time, but it is done so quickly that it is not noticeable to the human eye.

When drawing a line, the scanning beam starts at the left edge of the screen and proceeds to the right edge, shooting electrons the whole time. Since the beam has a finite amount of time it can spend drawing one line, the beam will seemingly have to move faster to cover more area on a larger screen. Thus the problem of trying to measure horizontal distances is further complicated by the fact that different scanning beams not only travel different distances, but also at different rates. Our unit of measurement cannot really be a distance; it must be a unit of time. The hint I gave a moment ago was that the scanning beam has a certain amount of time it can spend on one scan line. How fast or how far the beam travels is insignificant.

Understanding Color Clock

The fact that our unit of measurement is based on time explains the word *clock* in the term *color clock*. A color clock is the amount of time the computer needs in order to sufficiently change the frequency of the signal it generates so as to produce a different color. What a mouthful! This is my own personal definition; it has worked for me, but some people may not agree with it. Here's another definition. A scan line is the horizontal path of the scanning beam from the left edge of the screen to the right edge.

Scan lines extend horizontally across the screen, but it takes a lot of them stacked vertically to fill up the screen from top to bottom. Therefore, horizontal resolution is usually expressed in terms of color clocks while vertical resolution is expressed in scan lines. Of course, on different television sets the actual lengths will differ, but the resolution horizontally to vertically is always proportionate. It turns out that, on any screen, one color clock appears to be equal in length to two scan lines.

Now we have to get even more technical for a moment. The scanning beam starts at the upper left corner of the screen

and travels horizontally to the right. By the time it hits the right edge it has drawn one scan line that is 228 color clocks wide. The beam then shuts off for a short period while it returns to the left edge, only one scan line lower. This period is called the "horizontal blank" for obvious reasons. The beam then turns on again and starts drawing the next scan line. This sequence of drawing scan lines continues 262 times. At that point, the scanning beam, at the lower right corner of the screen, shuts off and returns to the upper left corner of the screen during a period known as the (guess what!) "vertical blank."

This whole process of drawing 262 scan lines, each of 228 color clocks, plus the blanking periods, constitutes one "frame." The television draws 60 of these frames every second, because your home power line is 60 Hz (cycles). The name given to this display method is "raster scan." The fact that your Atari follows a broadcast standard referred to as "NTSC" makes it one of the few home computers that can be video taped without special equipment.

Just because the scanning beam generates all those scan lines and color clocks doesn't mean that the computer is generating that much display data. Even if the computer did, you wouldn't see the whole image since most television sets display a little less than 200 scan lines of about 170 color clocks. The part where the true picture exists is called the playfield, and now it's time for another definition.

Playfields And Mode Lines

The playfield is the portion of each scan line for which data read from memory can produce colors and luminances. The background exists at the ends of each scan line; the playfield is in the middle. From the viewpoint of one frame, the playfield appears as a rectangular region which extends to the sides of the screen.

Two things control the size of this playfield area. The height in scan line is controlled by the display list as you will see in a moment. Recall that the width in color clocks is set by the DMA control register of the ANTIC.

SDMCTL	$022F	559	shadow
DMACTL	$D400	54272	hardware
D5	1 display list DMA enable 0 display list DMA disable		

D1,D0	00	playfield DMA disable (no playfield)
	01	narrow playfield (128 color clocks)
	10	standard playfield (160 color clocks)
	11	wide playfield (192 color clocks)

The operating system screen handler always uses a standard width playfield. The advantage of the narrow playfield is that less DMA is required, so programs execute faster. Unfortunately, the screen handler routines do not work properly when the playfield width is other than the standard. The wide playfield generates more data than the television can display; its uses are rather limited. It's even possible to turn off the playfield completely, in which case ANTIC fills the screen with scan lines of the background color. As will be shown in a moment, the playfield also requires a "display list," so bit five must be set for any playfield type to be generated.

Remember that a byte is made up of eight binary "bits." If playfield and display list DMA is enabled, bits may be read from the computer memory during the course of one scan line. The bit pattern determines the frequency and intensity changes of the scanning beam, with the result being different colors/luminances. The same bit pattern may be repeated for several scan lines. And the bit pattern can be interpreted in different ways. This leads us to yet another definition:

A mode line is a contiguous group of scan lines for which display memory is read only once.

There are two main types of mode lines. In direct memory map modes, the bit pattern produces the same image on each scan line. Text modes are a more complicated mode type which use a character set.

The ANTIC knows how to handle 14 different kinds of mode lines. Each mode line corresponds to a different method for interpreting a bit pattern. A full screen graphics mode is actually just a series of identical mode lines.

The display list is merely a sequence of bytes in memory that, among other things, tells ANTIC the proper sequence of mode lines for one screen.

Whenever the screen is opened (accomplished in Atari BASIC with the GRAPHICS statement), the screen handler establishes a display list of many mode lines to produce a screen of the desired mode. Modes can be mixed by manually changing the display list. Display lists produced by the screen handler always contain the proper number of mode lines for exactly 192

scan lines of playfield. Altering the display list can affect the total number of scan lines, which is how the vertical size of the playfield is controlled.

The display list also has other functions, such as control of fine scrolling, horizontal blank interrupts, and loading the memory scan counter of the ANTIC so it knows where to start reading memory.

A mode line divided into several parts forms pixels, which are single plotting points somewhere within the playfield area. A pixel's vertical resolution is the same as the mode line in which it is displayed, so there can be just as many pixels vertically as mode lines in the display list. The number of color clocks over which one pixel is spread is also determined by the mode line. Here is a little chart to show you the pixel size for the primary mapping modes:

MODE	COLOR CLOCKS	SCAN LINES	RESOLUTION (full/split screens)
3	4	8	40 by 24/20
4,5	2	4	80 by 48/40
6,7	1	2	160 by 96/80

Note that each time the width of a pixel is reduced, its height also decreases, so a single pixel appears to be square in shape regardless of the graphics mode.

Some Observations About Memory

Now to talk about memory. In the one-color modes, one pixel is represented in memory by one bit. If the bit is on, playfield zero shows. If the bit is off, the background shows. Modes four and six are the one-color modes. For more color, modes three, five, and seven allow three colors. The trade-off is that a single bit is no longer sufficient. Two bits, a pair, are required. The total value of the two bits selects either one of three playfields or the background:

BIT PATTERN	COLOR	PLAYFIELD TYPE
00	0	background
01	1	playfield zero
10	2	playfield one
11	3	playfield two

Playfield zero is the same thing as COLOR 1 in Atari BASIC. Playfield one is really COLOR 2, and so on, with COLOR 0 being the background.

Although modes four and five both have the same resolution, or pixel size on the screen, mode five will require twice as much memory. In the lower resolution modes which require little memory in the first place, the additional memory needed is rather insignificant. You might have noticed that mode three had no single color counterpart. Consider that in a 48K system it is possible to have about 150 different mode 3 screens in memory simultaneously. The chip designers probably decided it wasn't worth the effort or memory savings to provide a one-color mode with such low resolution.

Therefore, the size of a pixel on the screen is determined by two things: how many scan lines high, and how many color clocks wide. The amount of memory required for a mode is also determined by two things: how many separate pixels to one mode line, and how many color possibilities per pixel. The only real connection between pixel size on the screen and size in memory is that bigger pixels fill up a screen faster, so there are fewer of them, and less memory is needed.

Now, three colors means two bits must be used. Does that mean we are always stuck with only three colors which can't be changed? No. The CTIA is capable of generating 128 color/luminance variations. It can produce 16 different colors, each in eight different degrees of luminance. But 128 possibilities means seven bits would be required, and, in most cases, seven bits per pixel is simply not feasible. There is a limit to how much memory can be devoted to a screen. The solution to this problem is a sort of compromise, but it also presents some powerful and flexible advantages, too. The solution is to use *color indirection.*

Part II:
Atari Video Graphics And The New GTIA

Craig Chamberlain

How to get 256 colors out of your Atari. The previous article in this three-part series opened with a discussion of Atari graphics. Part II examines techniques involving color indirection *and looks at the new GTIA chip in detail.*

Next, this series concludes with several programs which put GTIA through its paces.

Using Color Indirection

With color indirection, the number of different playfields is limited according to the number of bits per pixel, but the actual color/luminance of each playfield can be one of the 128 possibilities. The data bits are used as an index or offset into playfield color registers:

```
COLOR0   $02C4   708
    playfield zero color register
COLOR1   $02C5   709
    playfield one
COLOR2   $02C6   710
    playfield two (used in modes 0 and 8)
COLOR3   $02C7   711
    playfield three (used in color text modes)
COLOR4   $02C8   712
    background color register
```

These playfield color registers use seven bits to select the color and luminance, as follows:

```
D7,D6,D5,D4      color
   D3,D2,D1      luminance
         D0      not used
```

BITS	VALUE	COLOR
0000	0	gray (no color)
0001	1	light orange
0010	2	orange
0011	3	red orange
0100	4	pink
0101	5	purple
0110	6	purple blue
0111	7	blue
1000	8	blue
1001	9	light blue
1010	10	turquoise
1011	11	blue green
1100	12	green
1101	13	yellow green
1110	14	orange green
1111	15	light orange

Atari BASIC allows you to select a playfield color to draw in by using the COLOR statement. The color register that corresponds to that playfield can be changed by using SETCOLOR.

Color indirection is a tool that should not be overlooked. It is possible to draw a detailed figure on the screen with one playfield, and then change the color of the entire figure with just one command. For example, a printed message can flash in colors to attract attention. A "glowing" effect can be created by rapidly changing the luminance of a playfield while maintaining the same color. Or, the playfield colors can all be set to the same color/luminance as the background. Figures drawn will not appear until the playfield color registers are changed. By changing the registers one at a time, an animation effect can be created. Color indirection may still not solve the problem of having many colors on the screen at the same time, but it does afford possibilities that otherwise would be difficult to achieve.

In special instances, playfield color registers can be changed during the horizontal blank, in which case all 128 color variations can be shown in one frame. This requires the use of machine language and still does not solve the problem of many colors on one scan line. Fortunately, experience has shown that, for many applications, three playfield colors will be sufficient.

Multiple Colors

Nevertheless, there are times when many colors would be desirable. This is where the GTIA steps in. It should now be apparent that 16 colors will require four bits per pixel. This is very expensive in terms of memory, so either pixel size or display memory will have to increase. Because ANTIC has a limit on how much memory it can access during one horizontal scan line, we have a limit on how much memory can be devoted to a screen. Therefore, resolution will have to suffer.

Before we see what the memory limit is, we should mention the two modes which are exceptions to the above rules. Three things distinguish modes zero and eight from the normal modes. Each pixel is a half color clock wide; a side effect of this is artifacting. The background color now becomes the border, and the main part of the screen is filled with playfield two. Finally, since the whole screen is now playfield two, the bit no longer tells which playfield to use, but which *luminance* to use.

MODE	BIT	LUMINANCE REGISTER
0,8	1	playfield one
0,8	2	playfield two (no image)

The color part of playfield one is ignored; only the luminance data is used. If the luminance values of playfields one and two are the same, the writing disappears. Modes zero and eight use this special "half color clock, one playfield color, two brightness" arrangement. Both modes have 320 distinct points of light horizontally and have single scan line resolution. The only difference between mode zero and mode eight is that the first is a text mode and the second is a direct mapping mode. Mode zero uses a character set and thereby saves memory; about 1K is required for this mode. Mode eight doesn't use a character set, and requires approximately 8K. That is our display memory limit. The Atari 400/800 is not capable of doing DMA to much more memory than the memory represented by one television frame.

Since the "half color clock, one color, two brightness" mode is used by graphics modes zero and eight, all the GTIA really does is provide three variations on this mode. They all use the maximum memory arrangement used by mode eight, so each of the three new modes requires 8K. All of the new modes use four-bit pixels, so the horizontal resolution goes from 320 (half color clock) to 80 (two color clock, as in modes four and five).

Therefore, the resolution for all three new modes is 80 by 192, for a total of 15,360 points. One side effect of changing only the horizontal resolution is that the pixels are no longer square.

The ANTIC instruction register mode number for the maximum memory mode (the number you will find in the display list) is $0F, or decimal 15. It is important to understand that this number indicates not only mode eight, but also nine, ten, and eleven as well. In fact, the display list for any one of these modes is identical to the display list for any of the others.

Selecting Modes With PRIOR

How then does ANTIC know which of the four is the desired mode? The answer is that ANTIC neither knows nor cares; no matter which mode is being used, ANTIC still has to do the same work of fetching memory. It's the GTIA that processes the video signal; somehow the chip must be told which of the four modes is wanted. The GTIA hardware register PRIOR does exactly that.

GPRIOR	$026F	623	shadow
PRIOR	$D01B	53275	hardware

The two most significant bits (bits six and seven) of this register are the GTIA special mode select bits. Here's how they are set.

MODE	BITS	HEX	DECIMAL
8	00	00	0
9	01	40	64
10	10	80	128
11	11	C0	192

For example, it is possible to switch from any one of the four modes to another simply by changing the values of the two select bits.

Other bits in GPRIOR serve different functions, so care must be taken not to alter them. These other bits allow multicolor players (blending on overlap), set all missiles to the color of playfield three to form a fifth player, and establish player/missile and playfield priorities. See the *Hardware Manual* for further information.

Now that we know how the three new modes are similar, let's find out how they are different.

Mode 11 is the one-luminance, 16-color mode. The overall luminance is set by the background color, which, for this mode,

defaults to a luminance of six, rather than the usual zero. It is now easy to draw rather finely detailed shapes in several colors without having to fool around with the display list and machine code interrupt routines. The thing I am especially excited about is going to make Apple owners envious. The Apple has a 16-color mode with resolution of 40 by 48, called the "lo res" mode. The Atari now has a 16-color mode, but the resolution is eight times greater than the Apple's.

Sixteen colors do present a problem, though, since the GTIA has only four playfield color registers. Therefore, mode 11 does not allow color indirection. The color on the screen is determined directly by the bit data stored in memory, according to the chart given earlier in the section on color indirection. The values in the four color/luminance registers are ignored. Some may consider this a disadvantage, but there is a benefit too. Just as the playfield color registers are not used, neither are the player/missile color registers used, so by using players it is possible to have 21 colors on the screen at the same time, without using display list interrupts or other tricks.

Producing 256 Colors

Mode nine is the one color, 16-luminance mode. This mode will be used to create some excellent three-dimensional effects and digitized pictures. The 16 luminances, when stacked vertically by the scan line with each line having the next brightest luminance, blend so well that it is very difficult to see the division from one to the other. The main color is set by the background color. Weird things happen when you change the luminance of the background. Another nice fact is that having 16 main colors with 16 luminance variations means that the Atari is capable of producing 256 colors.

One advanced application for mode nine is the display of digitized pictures. Digitization is a process by which a normal television picture, such as from a station or video recorder, can be analyzed and divided into different luminances. That information can be sent to the computer and stored on disk for later display. Mode nine, with 16 luminances and rather high resolution, is able to reproduce such pictures with impressive quality. Thus far we have seen only four digitized pictures. They were apparently made by some people at Atari, and two of the pictures were, uh, for mature viewers only. Standing from a short distance, however, it is very difficult to tell if any of these

pictures is computer generated or not. I have never seen such quality on any other computer in the 400/800 price range without expensive additional equipment.

Mode ten is a cross between the other two modes; it allows eight colors plus the background, each with its own luminance, as in the primary modes. Unlike the other two modes, this one allows color indirection, so it uses the playfield and player/missile registers for color/luminance information. This chart shows how data values correspond with playfield registers.

BITS	VALUE	REGISTER	PLAYFIELD
0000	0	704	PCOLR0
0001	1	705	PCOLR1
0010	2	706	PCOLR2
0011	3	707	PCOLR3
0100	4	708	COLOR0
0101	5	709	COLOR1
0110	6	710	COLOR2
0111	7	711	COLOR3
1000	8	712	COLOR4
1001	9	712	COLOR4
1010	10	712	COLOR4
1011	11	712	COLOR4
1100	12	708	COLOR0
1101	13	709	COLOR1
1110	14	710	COLOR2
1111	15	711	COLOR3

Only nine of the 16 possible data values correspond to different playfields. Data values greater than eight just repeat playfields. For some reason, the background color is no longer set by COLOR4, but instead by PCOLR0. The Atari BASIC statement SETCOLOR can't be used to change the player/missile color registers, so the equivalent POKE must be used. For any register, the data part of the POKE is the color choice number multiplied by 16, plus the luminance (refer to earlier chart).

The power of indirection is magnified when eight main drawing colors can be used. This mode is very useful for creating motion effects. With nine color/luminances and color indirection, mode ten may prove to be the most versatile of the three new modes.

Compatibility Between CTIA And GTIA

Remember that the GTIA only controls how the display is generated, so all programs written for the CTIA should run on a GTIA machine in the same way. There can be no such thing as

incompatibility. We have, however, come across one discrepancy between the CTIA and GTIA. The video signal generated by the GTIA is shifted one half color clock, so colors produced by artifacting, such as in *POOL 1.5* or *Jawbreakers*, will be different. That is just a minor visual difference; the important thing is that all software should be entirely compatible. Of course, you cannot expect a CTIA to generate these three new modes, but again the conflict is the display, not the program.

Because of the half color clock shift, it is now possible for players and playfields to overlap perfectly, whereas with the CTIA they didn't.

There are some cases where software will not run on GTIA machines. This is due to the fact that some of the new computers with the GTIA also have a revised (no bugs) operating system in them. Atari has made very clear which memory locations and vectors are permanent and protected from any revisions. If a program does not run on a GTIA machine, it is the software's fault because illegal entry points were used.

One other conflict has appeared which really surprised me. We have discovered that a few programs written on CTIA machines carelessly set the GTIA special mode select bits of GPRIOR for no purpose. Since these two bits do nothing on the CTIA, there was no problem. But there was also no reason to involve them. When the same programs are run on GTIA, the accidental bit settings affect the display, even though modes nine, ten, and eleven are not used. The function of those two bits has not been a secret. I figured out their function in July 1981, when I read the OS source listing before I bought my Atari 800. The *Hardware Manual* has described the three "new" modes in Appendix H ever since the manual was released.

No Text Window

There is a difference between the normal modes and the three new modes – the three new ones do not allow split screen (text window at bottom) configurations. If you remember how modes eight and zero are related, you should understand why. The mode used in the text windows is mode zero, which follows the special "half color clock, one color, two luminances" arrangement. As stated above, having the mode select bits in GPRIOR set for a mode greater than eight causes mode zero to act funny. A split screen would be possible only if a display list interrupt were inserted just before the text window area. The

interrupt routine would have to reset to zero the mode select bits in the hardware register PRIOR, not the shadow register. The hardware register will then be reset to the value of GPRIOR during the vertical blank service routine.

The three new modes seem to handle player/missile to playfield collisions a little differently. In modes zero and eight, a playfield two collision is flagged when a player or missile hits a pixel whose luminance is controlled by COLOR1 rather than the COLOR2 for the main playfield. From what I have been able to tell thus far, there is no kind of playfield collision at all in modes nine and eleven. Mode ten collisions work only for playfield colors that correspond to the usual playfield registers (COLOR1 through COLOR3). Also, the fact that the background in this mode is set by PCOLR0 affects the priority of players and playfields in some cases. In priority, mode ten playfield colors PCOLR0 through PCOLR3 behave like players.

The GTIA still allows only eight luminances on the normal modes.

All new Atari computers are being shipped with the GTIA at no extra cost. The CTIA is no longer being produced. The new machines with the GTIA have little yellow or white stickers that have the letter "G" on them. Those of us who have older machines with the CTIA can replace it with a GTIA. The part number is C014805.

If you want to do it yourself, it will be a simple matter to replace the CTIA. The CTIA is on the CPU card that plugs into the motherboard inside the Atari case. It's not soldered in, so the replacement operation should take only 30 minutes if you have taken your computer apart before. Instructions are supplied with the chip. In the meantime, if you don't have the GTIA, don't fret. It will be a while before much software requiring the chip is available.

Do You Already Have The GTIA?

If you want to quickly see if your computer has a GTIA, try this: POKE 623,64 (while in the default mode, zero).

If you have the GTIA, the screen will go black. Otherwise, there will be no change and you'll know you've got the CTIA. If you have the GTIA and want to see 16 colors, try this:

```
10 GRAPHICS 11
20 FOR K=0 TO 79
30 COLOR K
40 PLOT K,0
50 DRAWTO K,191
60 NEXT K
70 GOTO 70
```

Part III:

Atari Video Graphics And The New GTIA

Craig Chamberlain

In this conclusion of the three-part series, several demonstration programs teach the concepts of (and show off) the new Atari GTIA graphics chip.

Welcome back to our discussion of Atari playfield graphics and the exciting new GTIA chip. In Parts I and II I presented definitions of various terms related to graphics, explained the normal graphics modes, and then introduced the three new modes provided by the GTIA. Specifically, these new modes are:

MODE	DESCRIPTION
9	16 shades of one color
10	8 indirected colors
11	16 colors (one luminance)

Here are several programs in Atari BASIC to demonstrate how these new modes might be put to use. But first, let's tie up a few loose ends from the previous articles.

We used a standard method to show bit designations in the first parts of this article. If you are not familiar with this convention, here's how it works. Any given memory location or hardware address consists of one byte made up of eight binary units called bits. These bits are numbered zero to seven and are frequently shown as D0, D1, D7, etc. Individually, each bit can have two values, zero or one, but from the viewpoint of a byte, they take on quite different values known as "powers of two." For example, D3 means "two to the power of three," which also means "the number two used as a factor three times." Two times two times two is eight, so if we wanted to turn on only bit three in a given hardware register, we would POKE it with an eight. If we want to turn several bits on, we must add all the proper values together.

BIT	VALUE
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Mode 11 can be invoked by turning on bits six and seven of GPRIOR, location 623 (decimal). Thus we would POKE 623 with 64 + 128, which is 192. This brief explanation should help you deal with the memory locations and hardware registers described in the previous articles. Now for a review of the primary graphics statements of Atari BASIC and some special notes about the GTIA.

Graphics Statements

GRAPHICS aexp

This statement is the same as OPEN #6, 12 + 16,aexp, "S:", and tells the screen handler to open the screen to one of 12 modes. The number "aexp", which means "arithmetic expression," can range from zero to 11. Characteristics of these modes are explained in chapter nine of the *Atari BASIC Reference Manual.*

Some modes allow split screen configurations, which means that a text window appears at the bottom of the screen. Of course, mode zero does not allow a text window because mode zero is the text mode, although you can experiment with POKE 703,4. Modes one through eight do support text windows, and the only way to get a full screen (no text window) in one of these modes is to add 16 to aexp in the GRAPHICS statement. When using a full screen mode, Atari BASIC forces a mode zero if it has to print normal text. It is impossible to use these full screens in the immediate programming mode because the "READY" prompt forces the mode zero screen.

Due to technical reasons explained in the previous article, modes nine, ten, and eleven do not normally allow text windows. You can fool the operating system into giving you one of these modes with a text window by asking for mode 8 and then doing a couple of POKE statements, like this:

MODE	POKES
9	POKE 87,9: POKE 623,64
10	POKE 87,10: POKE 623,128
11	POKE 87,11: POKE 623,192

Location 87, known as DINDEX, tells the operating system the current mode and is used in the computation of row and column addresses for plotting, so any number from nine to eleven will give the same results. Unfortunately, the text window obtained by this method looks weird. The only way to get a real text window is to use a display list interrupt, discussed later.

If you add 32 to aexp, the screen will not be cleared when the new mode is requested.

Finally, the CTIA and GTIA support five other modes which the operating system does not recognize. They are the eight by ten matrix character version of mode zero, the multicolor text character modes, and the single scan line versions of modes six and seven, for 160 by 192 plotting in one or three colors. The only way to access these modes is to write a custom display list.

COLOR aexp

This specifies the playfield that will be used for PLOT and DRAWTO statements, until changed by another COLOR statement. It does not in any way change any of the color/luminance registers for the various playfields! The range of aexp depends on the number of different playfields available in the current graphics mode. This still holds true for the new GTIA modes. For example, a COLOR 2 in mode 9 means that future plotted points will be rather dark, whereas bright lines will be drawn after a COLOR 12.

In mode 11, COLOR 5 chooses a purple color, as indicated by the chart in part one of this article. For all modes, COLOR 0 (zero) is the background or "erasing" color. Normally, the operating system wants you to specify the playfield each time you write to the screen, but Atari BASIC automatically tells the operating system which playfield you have chosen every time you use PLOT or DRAWTO. Incidentally, the data part of the COLOR statement is stored in memory location 200 (decimal), but I would not recommend using that.

One other note. To be technically accurate, COLOR 1 corresponds to playfield zero, COLOR 2 means playfield one, and so on.

POSITION aexp1,aexp2
This statement moves the graphics cursor to the location on the screen designated by the two numbers, according to the Cartesian coordinate system. No range checking is done.

PLOT aexp1,aexp2
This is the same as POSITION aexp1,aexp2: PUT #6, color where "color" is the playfield type chosen by the most recent COLOR statement. You will get an error number 141 if you try to PLOT outside the bounds of the screen. All three new modes have resolution of 80 by 192.

DRAWTO aexp1,aexp2
Essentially, this is the same as PLOT except that a line is drawn from the most recently plotted point to the new point indicated by aexp1 and aexp2. You can also do this with an XIO 17,#6,0, 0,"S:". See page 54 of the *Atari BASIC Reference Manual* to see how XIO 18 can be used to fill areas with a playfield.

LOCATE aexp1,aexp2,avar
I don't know why, but nobody seems to know about this statement. It could be considered the reverse of PLOT. Instead of putting a playfield point at a certain location on the screen, this statement returns to you, in the arithmetic variable "avar", the playfield number of the point at location aexp1,aexp2. This playfield number will be the same as the value of COLOR that was in effect when the point was plotted. LOCATE is actually quite handy. It is very useful in games where collisions occur between differently colored players, but it has many other applications, too. LOCATE is the same as POSITION aexp1,aexp2: GET #6,avar.

SETCOLOR aexp1,aexp2,aexp3
This is the statement which changes the color and luminance of a playfield register. The number aexp1 designates which playfield register is being changed, and is related to the number in the COLOR statement in the following way:

COLOR	SETCOLOR (playfield number)
1	0
2	1 (also used for luminance in modes zero and eight)
3	2
–	3 (used only in four-color text modes one and two)
0	4 (background, or border in modes zero and eight)

The value for aexp2 is a number from zero to 15 which specifies one of 16 colors. See the chart in part one of this article, or on page 50 of the *Atari BASIC Reference Manual,* to find which numbers go with which colors. The luminance is chosen by aexp3, which can range from zero to 15, but only eight true luminances can be selected. A value of zero here gives the same luminance as one, two the same as three, and so on. The larger the number, the greater the luminance.

Remember that modes 9 and 11 do not use color indirection or the playfield registers, so SETCOLOR has little use in these modes. It can be used to set the background color/luminance in these two modes, but that's about it.

Now for mode 10. This mode uses the player/missile color/luminance registers, which cannot be accessed using SETCOLOR. An equivalent POKE statement must be used instead. The location to POKE is similar to the aexp1 of SETCOLOR. The shadows of the playfield registers run from locations 708 (decimal) to 712. The value to POKE contains the color and luminance information and is a combination of aexp2 and aexp3. This value is the sum of 16 times the color number, plus the luminance. In effect, SETCOLOR X,Y,Z will do the same as POKE 708 + X, 16*Y + Z. If you want to change the player/missile color/luminance registers, which run from locations 704 to 707, use the same procedure of multiplying the color by 16 and then adding the luminance. Refer to part one of this article for a chart that tells which COLOR numbers match with which registers.

Some Lively Demos

Now comes the good part, where the action is! If your Atari computer has a GTIA in it, here are some programs to show off the talents of this remarkable chip.

How to put 16 colors on the screen? It could be done in one line:

```
GRAPHICS 11: FOR K=0 TO 79: COLOR K: PLOT K,0:
  DRAWTO K,191: NEXT K: FOR K=0 TO 0 STEP 0: NEXT K
```

The endless loop is necessary to prevent Atari BASIC from printing a "READY" prompt which would force mode zero. To make the vertical color bands wider, change the COLOR K to COLOR K/5. To see 16 shades, change the GRAPHICS 11 to GRAPHICS 9.

A fancier way of showing 16 shades is found in Program 1. After drawing the shades, the background color is rotated through all 16 colors.

Program 3 randomly draws lines in 16 colors. You can make these colors appear darker or more pastel by changing the luminance of the background. Please note that mode 11 is the only mode in which the background is set by the operating system to a luminance of six. All other modes have backgrounds of color/luminance zero (black).

Program 2 demonstrates the color indirection capabilities of mode 10. Location 20 is the lowest counter of the realtime clock, so it is always changing. Continuously PEEKing this location and POKEing the value into a color register gives a nice "throbbing" color spectrum effect.

How about a doodling program that lets you draw in 16 colors using the joystick? Program 4 does this in only three lines of Atari BASIC code! Press the joystick trigger to change colors.

Program 5 is a really beautiful color kaleidoscope generator, considering it is only four lines long. It's not something you will spend hours watching, but it can produce some nice pictures. Try changing the K=I+J in the second line to K=I for a different picture. Or you can reverse the direction of the main loops, as in FOR I=31 TO 1 STEP -1. If you change the J loop (note that it starts at zero, FOR J=31 TO 0 STEP -1), you will also want to change the H loop (FOR H=1 TO 3 STEP 1).

To show 256 colors on the screen all at once, use Program 6. This program does not show the colors. Rather, it produces a single line which you can ENTER from disk or cassette. This single line performs all the magic. What is also neat about this program is that when you ENTER the line in, the program already in memory is untouched. If you examine Program 6, you will see that it writes a line to the chosen device, but the line has no line number in front of it. When you ENTER this line, it is the same as typing it in the immediate mode. When Program 6 asks for a device specification, respond with C: for cassette or D:filename for disk.

I included the assembly source code and Atari BASIC installation routine for a display list interrupt service routine (Program 8) that creates a text window on modes 9, 10, or 11. An interrupt is requested at the last mode line of the graphics mode portion of the screen. The service routine takes the value of

GPRIOR, sets the GTIA mode select bits to zero, and stores the result in PRIOR, the hardware register. PRIOR gets reset to the value of GPRIOR as part of the vertical blank service routine. The routine also stores a zero into the background hardware register. This was necessary to fix a conflict in mode 11. Setting the luminance in 712 also changes the border around the text window. But this "fix" created another problem in mode 10. For mode 10, change the fourteenth DATA element, which normally should be a zero, to be the same as the number POKEd into 704.

The service routine is written using relocatable code, so you can put the routine anyplace in memory simply by changing the assignment of ADDRES in the second line. It is currently set to start at the beginning of page six. The first three lines of Program 7 actually install the routine. The fourth line just draws a picture for purposes of demonstration. Notice the luminance change of the colors when 712 is POKEd.

My routine shares the problem of many display list interrupt service routines; keyboard clicks can affect the display. Obviously this routine is suitable only for programs that do not accept keyboard input (use the joystick or PEEK the hardware keycode register 764 directly) or use serial I/O (the vertical blank routine is abbreviated and PRIOR does not get reset).

Program 9 is a handy little routine which allows your Atari to test itself for a CTIA or GTIA chip – without your having to interpret screen colors, as other routines do. Now your programs can adapt themselves to either chip.

Program 1.

```
10 GRAPHICS 9:FOR K=1 TO 10 STEP 2:FO
   R J=0 TO 15:COLOR J:PLOT 0,K*16+J+
   1:DRAWTO 79,K*16+J+1
20 PLOT 0,K*16-J:DRAWTO 79,K*16-J:NEX
   T J:NEXT K
30 FOR K=1 TO 255 STEP 16:POKE 712,K:
   FOR J=1 TO 500:NEXT J:NEXT K:GOTO
   30
```

Program 2.

```
10 GRAPHICS 10:FOR K=705 TO 712:POKE
   K,12:NEXT K:FOR K=0 TO 79:COLOR (K
   +4)/10:PLOT K,0:DRAWTO K,191:NEXTK
```

```
20 FOR K=704 TO 712:FOR J=1 TO 300:PO
   KE K,PEEK(20):NEXT J:NEXT K:GOTO 2
   0
```

Program 3.

```
10 GRAPHICS 11:FOR K=1 TO 124:COLOR K
   :DRAWTO RND(1)*79,RND(1)*191:NEXT
   K:GOTO 10
```

Program 4.

```
10 GRAPHICS 11:DIM SX(15),SY(15):FOR
   K=5 TO 15:READ X,Y:SX(K)=X:SY(K)=Y
   :NEXT K:X=40:Y=96:COLOR 1
20 PLOT X,Y:X=X+SX(STICK(0)):X=X+(X<0
   )-(X>79):Y=Y+SY(STICK(0)):Y=Y+(Y<0
   )-(Y>191):IF STRIG(0) THEN 20
30 C=C+1-15*(C=15):COLOR C:GOTO 20:DA
   TA 1,1,1,-1,1,0,0,0,-1,1,-1,-1,-1,
   0,0,0,0,1,0,-1,0,0
```

Program 5.

```
10 GRAPHICS 10:FOR I=705 TO 712:POKE
   I,PEEK(53770):NEXT I:FOR I=1 TO 31
    STEP 1:C=C+1-9*(C=8)
20 POKE 704+C,PEEK(53770):FOR J=0 TO
   31 STEP 1:COLOR INT(RND(1)*15)+1:K
   =I+J:J3=J*3:K3=K*3:J8=J+8:J71=71-J
30 PLOT K+7,J3:DRAWTO K+7,191-J3:PLOT
    72-K,J3:DRAWTO 72-K,191-J3:FOR H=
   3 TO 1 STEP -1
40 PLOT J8,191+H-K3:DRAWTO J71,191+H-
   K3:PLOT J8,K3-H:DRAWTO J71,K3-H:NE
   XT H:NEXT J:NEXT I:POKE 77,0:GOTO
   10
```

Program 6.

```
100 IF PEEK(87) THEN GRAPHICS 0
105 ? CHR$(125):? "GTIA DEMONSTRATION
    ":?
110 ? "This program creates an ATASCI
    I file"
```

```
120 ? "for ATARI BASIC.  The file con
    sists"
130 ? "of one line which will produce
     256"
140 ? "colors on your screen if you"
150 ? "have a GTIA installed.":?
170 DIM S$(120):? "Please enter devic
    e specification."
180 INPUT S$:IF S$="" THEN 180
190 ? :TRAP 260:OPEN #1,8,0,S$
200 ? #1;"GR.9:F.K=0TO79:C.K/5:PL.K,0
    :DR.K,191:N.K:K=USR(ADR(";
210 PUT #1,34:FOR K=1 TO 15:READ P:PU
    T #1,P:NEXT K:PUT #1,34
220 DATA 173,11,212,10,229,20,41,240,
    141,26,208,208,243,240,241
230 ? #1;"))":CLOSE #1:? "File has be
    en written."
245 POSITION 2,19:? "ENTER ";CHR$(34)
    ;S$;CHR$(34)
250 POSITION 2,15:? "Now press the RE
    TURN key if"
255 ? "you want to ENTER the file.":N
    EW
260 STATUS #1,P:? "I/O ERROR ";P:END
```

Program 7.

```
10 POKE 54286,0:GRAPHICS 8:POKE 87,11
   :POKE 623,192:POKE PEEK(560)+256*P
   EEK(561)+166,143
20 ADDRES=1536:POKE 54286,64:FOR K=0
   TO 18:READ P:POKE ADDRES+K,P:NEXT
   K:P=INT(ADDRES/256):POKE 513,P
30 POKE 512,ADDRES-256*P:POKE 54286,1
   92:DATA 72,173,111,2,41,63,141,10,
   212,141,27,208,169,0,141,26,208,10
   4,64
40 FOR K=0 TO 159:COLOR K/10:PLOT 0,K
   :DRAWTO 79,K:NEXT K:POKE 712,6:STO
   P
```

Program 8.

```
0000          10            .PAGE

              11 ;
              12 ;necessary operating s
   ystem and hardware equates
              13 ;
026F          14 GPRIOR    =     $026F
            ;GTIA priority control (sha
   dow)
D01A          15 COLBK     =     $D01A
        ;background color register
D01B          16 PRIOR     =     $D01B
        ;GTIA priority control (hardwa
re)
D40A          17 WSYNC     =     $D40A
        ;horizontal blank synchronizat
ion
              18 ;
              19 ;
0000          20            *=    $0600
              21 ;
              22 ;this service routine
   for the display list interrupt
              23 ;can be placed anywher
   e in RAM, and was placed on page s
   ix
              24 ;only for purposes of
   demonstration
              25 ;
              26 ;begin interrupt servi
   ce routine code
              27 ;
              28 ;save contents of accu
   mulator
              29 ;
0600 48       30            PHA
              31 ;
              32 ;get the multicolor pl
   ayer, fifth player, and priority b
   its
```

```
                33 ;
0601 AD6F02     34          LDA  GPRIOR
                35 ;
                36 ;force the GTIA mode s
   elect bits to zero but save the ot
   her bits
0604 293F       37          AND  #$3F
                38 ;
                39 ;wait until next scan
   line for a nice clean change
                40 ;
0606 8D0AD4     41          STA  WSYNC
                42 ;
                43 ;change hardware regis
   ter until VBLANK
                44 ;
0609 8D1BD0     45          STA  PRIOR
                46 ;
                47 ;reset COLOR4 to zero
   (for modes 9 and 11)
060C A900       48          LDA  #$00
060E 8D1AD0     49          STA  COLBK
                50 ;
                51 ;restore accumulator
                52 ;
0611 68         53          PLA
                54 ;
                55 ;return from the displ
   ay list interrupt
                56 ;
0612 40         57          RTI
                58 ;
                59 ;end of interrupt serv
   ice routine
                60 ;
{L}
```

Program 9.

```
10 POKE 66,1:GRAPHICS 8:POKE 709,0:PO
   KE 710,0:POKE 66,0:POKE 623,64:POK
   E 53248,42:POKE 53261,3:PUT #6,1
20 POKE 53278,0:FOR K=1 TO 300:NEXT K
   :GRAPHICS 18:POKE 53248,0:POSITION
    8,5:? #6;CHR$(71-PEEK(53252));"TI
   A"
30 POKE 708,PEEK(20):GOTO 30
```

Protecting Memory For P/M And Character Sets

Fred Pinho

Redefined character sets and player/missile graphics both require protected memory. This article shows how to avoid memory conflicts.

Other articles in this book explain how to properly locate either a redefined character set or player/missile data. But what if you want to use both at the same time? You can't use the formulas given in these articles directly, because the two data sets will interfere with each other. To simplify matters, I've prepared the table below. It will allow you to position both your P/M and character data so that they won't clash.

Note that the two sets of data are stored in memory below the display list. Because of this, a couple of cautions are in order:

1. Be careful when changing graphics modes. If you go to a graphics mode requiring increased display memory, you could overwrite your P/M data. It's probably best to locate your data to accommodate the graphics mode with the largest memory requirement. To help, I've included the memory requirements for each graphics mode.

2. You have to watch your BASIC program to insure that it doesn't expand into your data-storage area.

In your reading, you'll come across other methods of storing these data sets (above a lowered RAMTOP, in a string, etc.). However, these methods have some serious limitations of their own. The method given here is straightforward and easy to trouble-shoot.

The table gives the offset from RAMTOP (in pages) needed

to properly locate each data set. Note that a page is a fancy way of describing a block of 256 data bytes. Also, the offset is subtracted from RAMTOP to get the proper memory location. I've given an example of use of the table below:

Desired: GRAPHICS 7
Single-Line Player/Missile Graphics
Redefined Full Character Set

Code:

```
10 PM=PEEK(106)-32:REM Calculate page
    setback for player/missile data
20 POKE 54279,PM:REM Set page number
   of PMBASE
30 PMBASE=256*PM:REM Calculate memory
    location of PMBASE
40 CHRSTART=256*(PM-4):REM Calculate
   page offset for new character set
50 POKE 756,CHRSTART:REM Set CHBAS to
    point to new character set locati
   on
```

From here, you can go on to implement the player/missile system and your redefined character set as described elsewhere in this book. May all your missiles be on target.

Simultaneous Positioning Of Player/Missile And Redefined Character Set Data in Memory.

	Memory required for Display Data + Display List		Player/Missile Data: Locate PMBASE at the indicated offset (in pages) from RAMTOP		Relocation of full character set beneath player/missile data: Relocate at the indicated offset (in pages) from RAMTOP	
Graphics mode	Total bytes	As whole pages	Double-line resolution	Single-line resolution	Character set with double-line P/M	Character set with single-line P/M
0	992	4	8	16	12	20
1	674	3	8	16	12	20
2	424	2	8	16	12	20
3	434	2	8	16	12	20
4	694	3	8	16	12	20
5	1174	5	12	16	16	20
6	2174	9	16	24	20	28
7	4190	17	24	32	28	36
8	8112	32	36	40	40	44

Notes:

A. General

1. All page calculations are to the nearest, highest whole page.
2. 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.

B. Player/Missile

Player/missile offsets are calculated by observing the following restrictions for the location of PMBASE.

	Double-Line resolution	Single-Line resolution
Offset from any other data	1K	2K
Boundary location for PMBASE	1K	2K

C. Character-Set

The full redefined character set must start on a 1K boundary (i.e., the first memory location must be a multiple of 1024).

Screen Save Routine

Joseph Trem

It certainly would be great if you could preserve for posterity those neat graphics designs you've learned to create. Here's one way.

The following utility routine can be appended to the end of your favorite drawing programs and will enable you to save those Rembrandts. A sample drawing is included at the beginning of this utility.

The Atari computer is fascinating indeed. The more you delve, the more intriguing it becomes. This program is based on three screen-related memory addresses – 87, 88, and 89. Location 87 contains the graphics mode presently in use. Type "GRAPHICS 7", then type "PRINT PEEK(87)". The computer will respond with "7". Locations 88 and 89 store the starting addresses of screen memory; 88 contains the low byte and 89 contains the high byte. Again, type "GRAPHICS 7", then type "PRINT PEEK(88) + PEEK(89)*256". This will return the memory starting address for GRAPHICS 7. Note that each computer may return a different number depending on the memory size of the machine. Now type "POKE(memory start), 255". This will light up one full byte at the top left corner of the screen (Figure 1).

Type "POKE(memory start + 40,255)", and this will light up the next full byte directly under the first byte. Knowing this, it is possible to keep track of every byte on the screen. There are 40 bytes horizontally and 80 bytes vertically in GRAPHICS 7. In the utility program, line 32240 locates the starting address of your picture. Lines 32125 and 32225 scan and set screen memory locations. You may adapt these lines to any graphics mode using the chart provided. For example, if you happen to be using GRAPHICS 5, change the "40*80" in those lines to "20*40". (See Figure 2.)

After running this program, you may want to append only the utility part to your favorite drawing program. Here's how to do it. First, make sure your drawing program does not exceed line 30999. Now type LIST"C:",31000,32240 or LIST

"D:filename", 31000,32240. This will save only lines 31000 through 32240. When completed, type NEW and load your drawing program. Now load your utility program back in. This is done by typing ENTER"C:" or ENTER "D:filename". This will append your utility to the end of any drawing program.

Some programs may have to be modified slightly, but with a little effort you may find it worth it. Run your program. Draw your masterpiece. When you are satisfied with your creation, press the BREAK key and type "GOTO 31000". This will initialize the save and load routine. Then sit back, relax, and surprise someone with a genuine work of art worthy only of the great masters.

10 Initializes SCREEN SAVE ROUTINE
40-195 Draws sample picture (e.g., space game playfield).
200 Reinitializes menu after drawing.
30000 Sets GR.2 and title.
31000 Opens IOCB for keyboard.
32000-32060 Prints menu. Gets keyboard input and directs to appropriate line.
32100-32103 Prints save menu. Gets keyboard input and directs to appropriate line.
32105-32208 Prints disk instructions.
32110-32210 Prints cassette instructions.
32200 Prints load menu.
32122 Stores graphics mode and color register data.
32125 Stores screen RAM data.
32222 Reads graphics mode and color register data and POKEs it into correct locations.
32225 Reads screen RAM data and POKEs into correct locations.
32240 Determines start address of screen.

Figure 1.

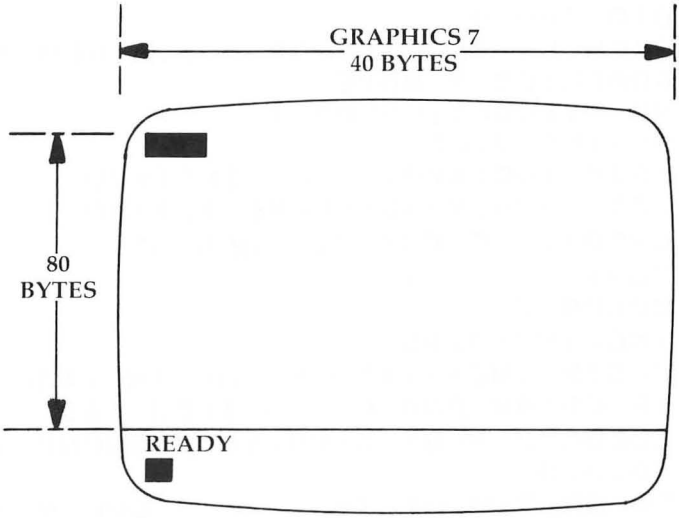


Figure 2.

	HORIZONTAL BYTES	VERTICAL BYTES
GR.8	40	160
GR.7	40	80
GR.6	20	80
GR.5	20	40
GR.4	10	40
GR.3	10	20
GR.2	20	10
GR.1	20	20
GR.0 (full screen)	40	24

```
5 REM * SCREEN SAVE ROUTINE *
8 REM * JOE TREM *
10 GOTO 30000
40 GRAPHICS 7:SETCOLOR 0,6,6:REM *SET
    GRAPHICS 7 MODE
60 INC=49:CO=1:COLOR 1
65 INC=INC+0.05
70 X=SIN(INC)*20:Y=COS(INC)*20
75 PLOT X+80,Y+35:SOUND 1,X+50,10,8
80 CO=CO+1:IF CO<130 THEN 65
100 CO=1
110 COLOR 3
130 INC=INC+0.05
140 X=SIN(INC+1)*30:Y=COS(INC)*30
145 IF CO>49 AND CO<70 THEN 160
150 COLOR 2:PLOT X+80,Y+35:SOUND 1,X+
    50,6,8
155 COLOR 3:PLOT X+81,Y+38:COLOR 1:PL
    OT X+79,Y+32
160 CO=CO+1:IF CO<130 THEN 110
170 COLOR 1:FOR X=1 TO 159:PLOT X,79:
    DRAWTO X,79-RND(0)*5:SOUND 1,X,10
    ,8:NEXT X
180 FOR X=1 TO 20:COLOR RND(0)*1+1:PL
    OT 40,40:DRAWTO RND(0)*10,RND(0)*
    10:SOUND 1,X+20,8,8:NEXT X
183 FOR X=120 TO 159:COLOR RND(0)*2+1
    :PLOT X,20:DRAWTO X,RND(0)*20:NEX
    T X
185 COLOR 2:PLOT 130,30:DRAWTO 130,24
    :DRAWTO 134,24:DRAWTO 134,30:PLOT
     130,27:DRAWTO 134,27
186 PLOT 142,25:DRAWTO 142,24:DRAWTO
    138,24:DRAWTO 138,30:DRAWTO 142,3
    0:DRAWTO 142,29
187 PLOT 150,24:DRAWTO 146,24:DRAWTO
    146,30:DRAWTO 150,30:PLOT 146,27:
    DRAWTO 149,27
190 COLOR 2:PLOT 0,0:DRAWTO 159,0:DRA
    WTO 159,79:DRAWTO 0,79:DRAWTO 0,0
195 COLOR 1:PLOT 30,70:PLOT 40,10:PLO
    T 140,30:PLOT 150,70:PLOT 105,35:
```

```
    SOUND 1,0,0,0
200 GOTO 32000
30000 GRAPHICS 2:? #6;" SCREEN SAVE R
      OUTINE"
31000 CLOSE #1:OPEN #1,4,0,"K:"
32000 SCRN=32240:GOSUB SCRN:POKE 752,
      1:? "            1...DRAW PICTURE
      "
32010 ? "            2...SAVE PICTURE"
32020 ? "            3...LOAD PICTURE"
32050 GET #1,A:IF A<49 OR A>51 THEN 3
      2050
32060 ON A-48 GOTO 40,32100,32200
32100 ? :? "  SAVE TO  1...CASSETTE?"
      :? "             2...DISK?"
32101 GET #1,A:IF A<49 OR A>50 THEN 3
      2101
32103 ON A-48 GOTO 32110,32105
32105 ? :? "   PLEASE INSERT DISKETTE
       AND PRESS       RETURN":GET #1,
      A:OPEN #2,8,0,"D:PICTURE":GOTO
      32120
32110 ? :? "PLEASE PLACE CLEAN TAPE I
      N RECORDER              AND PRESS
       RETURN"
32115 OPEN #2,8,0,"C:":REM *OPEN FILE
       TO SAVE
32120 ? :? "SIT BACK AND RELAX... SAV
      ING PICTURE"
32122 MODE=PEEK(87):PUT #2,MODE:FOR I
      =0 TO 4:COL=PEEK(708+I):PUT #2,
      COL:NEXT I
32125 FOR I=SCREEN TO SCREEN+(40*80)-
      1:LOC=PEEK(I):PUT #2,LOC:NEXT I
      :CLOSE #2
32130 GOTO 32000
32200 ? :? "  LOAD TO     1...CASSETTE
      ?":? "              2...DISK?"
32201 GET #1,A:IF A<49 OR A>50 THEN 3
      2201
32203 POKE 752,1
```

```
32205 ON A-48 GOTO 32210,32208
32208 ? :? "  PLEASE INSERT DISKETTE
      AND PRESS        RETURN":GET #1,A
      :OPEN #2,4,0,"D:PICTURE":GOTO 3
      2220
32210 ? " PLEASE INSERT TAPE AND PRES
      S RETURN"
32215 OPEN #2,4,0,"C:":REM *OPEN FILE
       TO LOAD
32220 ? :? :? "RELAX AND ENJOY... LOA
      DING PICTURE"
32222 GET #2,MODE:GRAPHICS MODE:GOSUB
       SCRN:FOR I=0 TO 4:GET #2,COL:P
      OKE 708+I,COL:NEXT I
32225 FOR I=SCREEN TO SCREEN+(40*80)-
      1:GET #2,LOC:POKE I,LOC:NEXT I:
      CLOSE #2
32230 GOTO 32000
32240 SCREEN=PEEK(88)+PEEK(89)*256:RE
      TURN
```

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, {A}.

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

alternate shapes data 185
alternating color bands 204-207
animated games 98-107
animation 11,91,99,108,126, 172-183,184-187
animation demo 143,152-153
ANTIC chip 110,166,208,209,211, 213,217,219
ANTIC display modes 102
artifacting 206-207
ASCII 16,92,99,113
Atari BASIC Reference Manual 3, 10,11,18,38,54,55,56,81,113, 203,225,227,228
Atari Hardware Manual 192,193, 219,221
ATASCII 11,42,54,57,58,81,91,92, 114,142-143
BASIC 3,18,25,26,33,62,64,78,79, 80,81,82,83,84,91,98,99, 108-109,118,132,184,185,195, 209,212,216 (see also GRAPHICS commands)
BASIC A+ 133
BREAK 240
Central Processing Unit (CPU) 129,222
character graphics 11,111,116
character registers 78
character set 77,82-83,85,121
    relocating 78-82,85
character storage 79-82
character string 43,131
collision registers 188-191,195
collisions
    players with players 188-189, 191
    players with playfields 189, 191
color clock 209,210-211,213
color indirection 215-216,229, 230-231
color numbers (table) 14
Color Register Default values (table) 14
color registers 21,215-216
COLOR statement 9,20,21,93, 109,129,130,189,190,216,227
colors, demo programs 228-235
control graphics 12
controls for P/M graphics 132-135
coordinates 8,10,13,99,100,114, 165,169,173-174,176,177
Creative Computing 142
"crosshair" 197-199
CTIA chip 4,9,110,188,208-209, 214,220-221,230
CTRL key 111,116,117,119,121
default color 7,194
delay loop 179
diagonal lines 206-207
digitized pictures 219-220
direct memory access (DMA) 166,168,192,209,210,211,212
display data 165,238
display image 174
display list 26,27-33,37-38,41,46, 80,82,91,131,212,213,236,238
display list interrupt 62,68,82, 83,87
display memory 29,30,41
DOS 94
drawing storage 172-180
figure manipulation 18,19
function codes 47
games
    animated 98-107,108-126
    Asteroids 98
    "Island Jumper" 156-157, 159-162

Jawbreaker 195,221
Pac-Man 3,195
Pong-type games 20-22
POOL 1.5 221
"Space Rocks" 98,100-107
Star Wars 111
Star Raiders 3
GPRIOR 219,221,230
GRAPHICS characters 4,53,54
GRAPHICS commands 4,25,56, 66,112,130
DRAWTO 8,9,10,16,99,110,130
LOCATE 10,11,12,21,99, 100,111,227
PLOT 8,9,10,16,99,110,130, 131,227
POSITION 10,11,12,44,110, 111,227
PRINT 10,11,39,44,110
graphics modes 13,25-36,80,95, 113,177,213,224,241
GRAPHICS 0 4,5,6,11,15,31, 38,44-45,46,56,59-60,88,130, 137,167 (see also text mode)
GRAPHICS 1 4,33,34,61,82, 83,84,112,113,115,116,117, 177,195
GRAPHICS 2 4,6,10,39,42, 61,82,113,195
GRAPHICS 3 4,5,7,16,20,33, 34,37-38,91,130
GRAPHICS 4 6
GRAPHICS 5 91,239
GRAPHICS 6 6,7,8,9,15,92, 94
GRAPHICS 7 9,38,91,99, 109-110,131,133,237,239
GRAPHICS 8 5,6,46,93,112, 131,167,203-207
GRAPHICS 11 196,223,225, 226,228
GRAPHICS statements 225-228
graphics string 16-18
GTIA chip 4,9,110,188,193,196, 208-235
hardware register (see PRIOR)
high resolution 203
horizontal blank 211
IF statements 65
image data 158
image memory 155
Instruction Register (IR) code 28,29
Internal Character Set 116
table 120
interrupts 163-167 (see also vertical blanks)
inverse characters 77,78
LMS command 29,30,34
location argument 156
luminance 7,210,214,217,219, 222,228
machine language routines 143, 154,175,184,
for animation 184-187
for character storage 86
memory 6,133,213, (see also RAM)
memory allocations 167-168, 173,174
memory locations 42,55,78, 79,100,109,118,132,139,156, 221
memory protection 236-238
missile registers 190
mixed modes (see text window)
mode lines 26-27,211-213
Moiré patterns 207
multiple color player enable 193, 196-197
number of colors 5,6-7,25
Operating System 26,27,28,29, 30,32,55,192
overlaps 196-197,199 (see also collision registers)
overscan 27
page six 81,93,166,173
pages 78,155,176-177
paging 55,56
player drawings 172-179,181
player image 156

player memory 136
player motion 142-143
player storage 173-176,178
player/missile (P/M) graphics 110,129-139,140-153
P/M memory locations 139
players 137,180
playfield graphics 129,130,134, 211-213,216
RAM positioning 138
playfield registers 220
PMBASE 138,167,174,176,180, 184,192
POKEs 31,132,175,185,189,193, 196,228
Pong-type games 20-22
PRINT #6 19,44,45
PRIOR 218,222,230
priority registers 192-199
programs
fast graphics 122-125
"Island Jumper" 159-162
Mixing Modes 0 and 8 (demo) 48-49
P/M Graphics Utility 144-151
Screen Save Routine 242-244
"Space Rocks" 102-107
TextPlot 95-97
pseudo-random number function 169
RAMTOP 55,78,167,236-237
Random Access Memory (RAM) 6,78,131,134,135,167,184, 185,192
raster scan 211
redefining character sets 53-61, 62,77,82
registers 7,132,188-191
character registers 78
collision registers 188-191,195
color registers 21,215-216
hardware register 218,222,230
Instruction Register 28,29
missile registers 190
playfield registers 220
priority registers 192-199
"shadow" register 77,78,83, 190,222
resolution 5,25,138,203,219,238
horizontal 5,210,217
single line 167
vertical 5-6,210,213
ROM 42,55,57,61,65,77,78,80,192
scan lines 26,29,210,213,217
screen limits 121
screen memory 38,93,113,203-207
screen save utility 239-244
scrolling 33,44
SETCOLOR statements 7,9,20, 21,65,129,130,131,169,216-228
"shadow" register 77,78,83,190, 222
string graphics 16
string manipulations 142
SuperFont 62-76
commands 63-65
text editor memory 38,39
text modes 4,46,77,88,91,92-94, 195,212,217,225-226
text window 5,6,26,33,38-39, 44-45,46,83,88,177,193, 221-222,225-226
user memory 55
USR function 46,81,99,143,154, 155,165,173,184
variables 31,91,102,111,117,118, 142,143
DL 31
SCREEN 111
for "Space Rocks" 102
vertical blank interrupt 164-167, 173-179,192
vertical blank P/M routine 183
vertical blank time 165
vertical blanks 77,83,172
vertical movement 164,166
vertical positioning 154-163, 164-167
assembly language representation 163

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!**

For Fastest Service,
Call Our **Toll-Free** US Order Line

800-334-0868

In NC call 919-275-9809

COMPUTE!
P.O. Box 5406
Greensboro, NC 27403

My Computer Is:
☐ PET ☐ Apple ☐ Atari ☐ VIC ☐ Other _____ ☐ Don't yet have one...

☐ $20.00 One Year US Subscription
☐ $36.00 Two Year US Subscription
☐ $54.00 Three Year US Subscription

Subscription rates outside the US:
☐ $25.00 Canada F=2
☐ $38.00 Europe/Air Delivery FI=3
☐ $48.00 Middle East, North Africa, Central America/Air Mail FI=5
☐ $88.00 South America, South Africa, Australasia/Air Mail FI=7
☐ $25.00 International Surface Mail (lengthy, unreliable delivery) FI=4,6,8

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

☐ Payment Enclosed ☐ VISA
☐ MasterCard ☐ American Express
Acc't. No. _____ Expires ____ / ____

08-6

COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868

In NC call 919-275-9809

Quantity	Title	Price	Total
_____	**The Beginner's Guide To Buying A Personal Computer** (Add $1.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$ 3.95**	_____
_____	**COMPUTE!'s First Book of Atari** (Add $2.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$12.95**	_____
_____	**Inside Atari DOS** (Add $2.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$19.95**	_____
_____	**COMPUTE!'s First Book of PET/CBM** (Add $2.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$12.95**	_____
_____	**Programming the PET/CBM** (Add $3.00 shipping and handling. Outside US add $9.00 air mail; $3.00 surface mail.)	**$24.95**	_____
_____	**Every Kid's First Book of Robots and Computers** (Add $1.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$ 4.95**	_____
_____	**COMPUTE!'s Second Book of Atari** (Add $2.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$12.95**	_____
_____	**COMPUTE!'s First Book of VIC** (Add $2.00 shipping and handling. Outside US add $4.00 air mail; $2.00 surface mail.)	**$12.95**	_____

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. _____ Expires ____/____

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.

08-6

ISBN 0-942386-08-6

$12.95 US