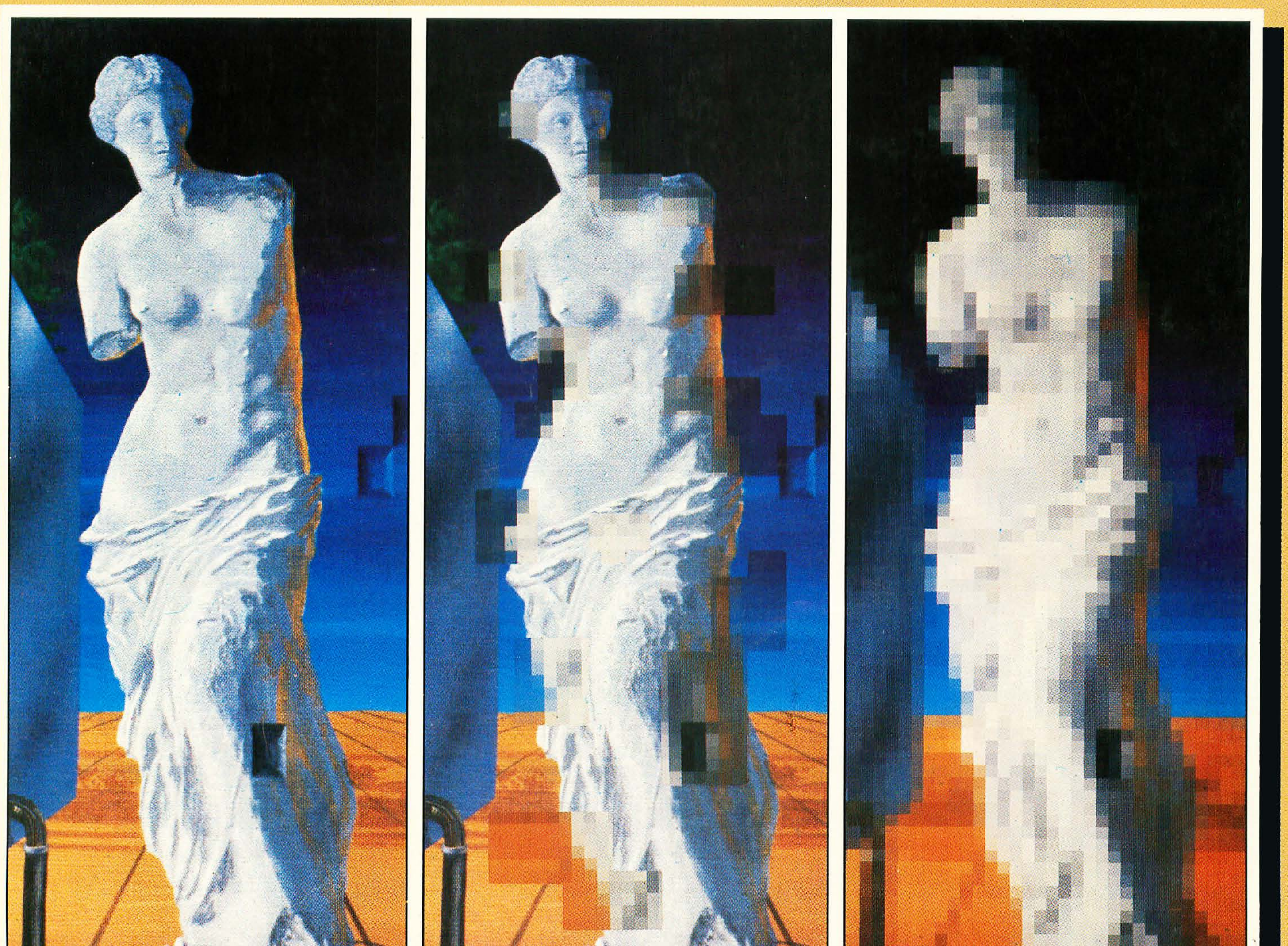
THE CREATIVE ATARI



Edited by David Small, Sandy Small and George Blank

The Creative Atari

The Creative Atari

Edited by: David Small, Sandy Small, and George Blank

Creative Computing Press
Morris Plains, New Jersey

David Small, Sandy Small, and George Blank are frequent contributors to *Creative Computing* magazine.

Unless otherwise noted, all work contained in this volume is that of the editors. Special thanks to John Anderson for technical assistance.

Library of Congress Number: 82-71997
ISBN 0-916688-34-8

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Creative Computing Press
39 E. Hanover Ave.
Morris Plains, NJ 07950

Preface

If you own an Atari computer, you should already be familiar with *Creative Computing* magazine. If you are not, than you have missed out on some of the best material ever published regarding the Atari 400, 800, and new 1200 XL computers. Our commitment to the Atari machine has been a steady one, and includes information you won't find in any owner's manual or technical reference sheets. The Atari tutorial and programming techniques offered in the pages of *Creative* over the past three years have been consistently presented in a way you can understand and build upon and much of this material is still available through no other source.

But fear not, newcomers to the fold, because this book is for you. It contains all the material you have missed, and more. And all of it concerns the Atari personal computer and only the Atari. Which, you will probably agree, is by far the best microcomputer in its class.

If you are not sure you agree, then you really need this book. It shows you in simple terms how to get the most from your machine in the realms of graphics, sound, memory, animation, disk storage, and a dozen other topics. We don't assume you are anything more than an enthusiastic beginner, and step-by-step examples are a hallmark of ours. We know that learning by doing is the best way to learn.

If you are already an old fan of *Creative Computing,* this book is still for you. In addition to giving you a single reference for all Atari material we've yet published, it includes new material, published now for the first time.

You won't find a single theme here, with all the articles winding neatly around it. What you will find is an Atari almanac, a meaty compendium, full of valuable, "hands-on" projects to keep you and your Atari busy for months and months to come.

In addition, you will find in-depth tutorials, product reviews, philosophical ramblings, insider's gossip, and insights into the impressive powers, as well as occasional weaknesses (and how to get around them) of the Atari machine.

Of course, for up to the minute Atari developments, tune in to *Creative Computing* every month!

John Anderson
Associate Editor
Creative Computing

Contents

Part I. Atari Graphics Tutorial

Part II. Graphics Tips

Part III. Hardware and Software

Part IV. Disk Drive Tutorial

Part V. User Programs

The Atari Machine

Ted Nelson

I first saw Atari's Mean Machine at an education-and-computers conference last September. A lot of pompous Educators had come to receive the word from some Foundation People about the blessings that small computers and videodisk were about to bring to them. I was there with Stuart Greene, an associate and filmmaker who also has a sense of what computer graphics ought to do.

Well, there was a keynote address from the highest Foundation Person, and good things were said; and then a wonderful thing happened.

Up got Ludwig Braun with his fierce mustache and apologetic manner, Lud Braun who has tried indefatigably for so long to arouse the educational establishment to the educational potential of simulation and little computers; up got he, at an Advent screen, and said he had a new machine to show us.

He turned on the Atari.

Here is what we experienced.

We are on a spaceship, cruising at near-light speeds. Stars are on the screen, but they part before us, moving smoothly *out from a common center* as we cleave the void. A low rumble—ship's noise or remanant Big Bang—accompanies our movement.

The pilot turns. The stars still move apart for us, but now the center of diverging motion has moved to another part of the screen. Stars pass each other—they must be the near ones—and we see that the display really shows us moving through stars in *three dimensions.*

PLANETS shoot by.

Enough of the slow stuff. Let's take this baby out for a spin.

Acceleration! The rumble rises in pitch and volume. The stars really start to fly apart. HYPERWARP ENGAGED, flashes a warning on the screen. Faster and faster shoot the stars, as from a Fourth-of-July sparkler, AND NOW THE SCREEN IS RED IN SUDDEN SILENCE, AND IT FLASHES "HYPER-WARP"!

And out again! There is roaring anew, and new stars split to let us pass, but we are slowing down now. The rumble lowers. We have gone halfway across the universe.

Stuart and I were shouting and cheering and clapping. I think I may have been on my feet with excitement. The Educators turned to stare at us. "What does this have to do with Education?" asked their faces. Guys, if you don't know, we can't tell you.

I've been in computer graphics for twenty years, and I lay awake night after night trying to understand how that Atari machine did what it did.

As I have always known the field, there are basically two kinds of computer graphics machines. The bit-map machines, the video type, have a fixed number of dot positions, and if you want to "move" a shape, you have to keep erasing it at one spot and re-writing it at the next. (The Apple computer's hi-res is of this type.) Either the movement is cyclically jerky, as your movement subroutine reaches different picture elements, or you have to prepare a "next frame" in a different area of core, which may be slow, and flip the new image to the screen when it's all ready. (The Apple allows this.)

Problems arise when a moving figure crosses a still figure; restoring the background after a moving shape has passed is a real problem. Preparing an unseen Next Frame that restores the background is again the solution, but that takes still more time.

Then there's the other kind of graphics machine, the Super kind—the "calligraphic" display—where points and lines are individually placed on a rasterless screen. Special hardware steps through a display list in core, putting each part of the picture where the program says. Each time the screen is refreshed, the points and lines can be moved individually as your program changes the screen positions specified by the display list. (Examples are the Picture System from Evans and Sutherland for $100K, or in the $15K ball-park, Imlac's PDS-4 and DEC's VT-11.)

But this, *this* new machine, was something else.

In a package under one thousand dollars, and using a conventional raster screen—a TV—the Atari computer was doing smooth motion in all directions at once, seemingly in 3D.

This had to mean, I reasoned, that there was some sort of a DMA readout from core (as in the calligraphic machines), in order to match the raster-timing demands of the TV screen. But then there would have to be some sort of address translator, allowing the element itself to remain on a display list in core, where its screen address could be changed between frames.

But then there would also have to be some list, corresponding to the picture arrangement on the screen, of where everything was in core.

It just didn't make sense.

Well, I know how it works now, dear reader, and I wish I could tell you. But, unfortunately, *Creative Computing,* as a software producer, has signed a non-disclosure agreement with Atari, so that anything I've learned through these channels I can't published. But aha, if I can find it out through *other* channels, says Dave Ahl in his Solomonic wisdom, then I can publish it. So I will be spying assiduously, dear reader, to find out what I already know so I can tell you about it. Ah, modern life.

The Atari machine is the most extraordinary computer graphics box ever made, and *Star Raiders* is its virtuoso demonstration game. Be not misled by the solidity of the *Star Raiders* capsule you must push into place; it is not hardware. It is a program.

Yes, friends, all the effects I have described—and many more indeed—can be programmed on the Atari.

There is just one problem.

They won't tell you how.

That's right. You can buy an Atari computer and they won't give you instructions on how to work it. Everything is under wraps. Oh, of course you can program the 6502 chip, that's in there, same as in the Apple. But that *other stuff,* those mysterious peek-and-poke locations that move the stars, and whatever else they do do, are a deep dark secret.

Now, I'm pretty sure that if you wanted to bring a case before the Federal Trade Commission, there's some statute saying you're entitled to get operating

Ted Nelson, 8631 Fairhaven, Apt. 109-13, San Antonio, Texas 78229.

instructions for whatever you buy. So if you want to make a federal case out of it, you can probably get the inside data in about three years for a quarter of a million dollars in legal costs. However, there's a faster way.

Wait.

The hacker's race is on. Who can figure it out first?

Even if nobody violates Atari's elaborate security, I'll wager that most or all of the secrets of the Atari machine will be out by the end of 1980—probably including secrets that the Atari people didn't know existed. Because there is nothing like a real challenge to delight a computer hacker, and this is a real challenge.

Now, there are all kinds of signs in the wind. For instance, one California company, advertising in these very pages, says they have a book on the Secrets of the Atari. Not to mention a disassembler that will ferret out even the deepest secrets of *Star Raiders.*

I called them about the book and they said well, it wasn't quite ready yet, and when I asked for galleys they alluded to how it wasn't quite written yet, but I'm sure it will be a very good book when it comes out, and that they won't be the only sources for the information. Because if there's one thing that makes the world go round it's gossip, especially juicy true gossip, like how to control horizontal scroll or interrupt on raster-line count (just to take fictitious examples).

An interesting question is why Atari is bothering to hide the information at all, and from whom. Is the information being hidden from the purchaser of the Atari computer? That would hardly seem proper, let alone sane. From rival hardware manufacturers? Fiddle de dee. The last thing any hardware rival would do would be to sink hundreds of grand in copying the Atari special chips. Anyone who has the temerity to design a computer always thinks he can do it better anyway. (One conceivable possibility is that Third-World Manufacturers might try to build imitation Ataris—as has been done for the TRS-80, but not the Apple. It seems a lot of effort for a far-fetched threat—especially considering the system price, which is an extraordinary value; it's hard to see how Taiwan or the Philippines could compete with it in price for several years. Perhaps the Atari folks are just that sure of their own infallibility that they worry about others horning in on a multi-million-unit market.)

Another interpretation is that the Atari people are trying to hobble potential software rivals. If nobody else knows how to get the hotshot effects, then the Atari guys have an advantage with their software, right? Again a strange notion. Since Atari makes the machines, why do they mind? (Anyway, Atari is being cooperative with independent software vendors, *provided* they don't tell how it works. So the whole thing is very mysterious.)

What It Can Do

The only way to explain fully what the Atari will do is to reveal its internal hardware structure. As explained above, that cannot happen here yet. However, there is a very simple way for you to study the capabilities of the Atari machine: that is to go to your local video-game arcade and play the Atari arcade games. Everything they do, the Atari computer will do. (I know of only one exception: the "Lunar Lander" Atari game, which uses vector graphics and is therefore incompatible.) Two very good examples for study, if you can find them, are "Basketball" and *"Star Raiders."*

(I regret that Sky Raiders is a shoot-em-up game, or, indeed, that our society has such a high regard for games where you get high scores for murdering lots of imaginary adversaries. It could be argued that Vietnam, the Body-Count War, was born in the arcades of yesterday, and that Star Trek games are setting us up for World War Three—but that's a different article. Anyway, consider that the effects you are seeing can be put to peaceful uses, like the teaching of physics and watching the flowers grow.)

Here are some things you should look at.

The way that the whole screen can be filled with shaded graphics, that is, pictures made out of colors or grey levels. (Colors are not much used in Atari arcade games, with some exceptions like the multi-car Speedway game. But the colors are just fine on the Atari computer.)

The way that pictures of small objects can move across this overall picture without disturbing it. (Examples: basketball players in the Basketball game; automobiles in the speedway game; the hook and ladder truck in the Fire Truck game, which can actually be driven across people's lawns and driveways, and through their houses, with very satisfying sound effects.)

Of all the Atari arcade games, the most portentous, in my opinion, is *Star Raiders.* This is a bombardier game in which you get points for destroying cities, factories and power lines. (Again, ignore the shoot-em-up aspect.) What you see is a continuing panorama unrolling below your bombsight: the aerial view of the countryside. The video monitor is mounted vertically, and the aerial view descends down the screen—sideways on the video.

In other words, what you are seeing is *horizontal scroll* of detailed graphics relative to the monitor.

Another feature that merits your close attention is the interaction between moving objects and the background. In the basketball game, for instance, not only do the two players move around in front of a full background picture; they also *block one another:* either the black player is in front of the white player, or vice versa. You may have an interesting time thinking about what hardware this implies.

Moving objects may also interact with the background picture. For instance, in the *"Star Raiders"* game, a bomb which is on target creates an explosion on the ground. This implies interesting interaction between the data about moving objects and the data about the background.

Well, Space Troopers, that's it for now. The Atari is like the human body — a terrific machine, but (a) they won't give you access to the documentation, and (b) I'd sure like to meet the guy that designed it. □

Part I
Atari Graphics Tutorial

How a TV Works

In order to understand how the Atari does its work, we need to know how a TV set works. (If you already know all about raster scan and similar concepts, you can probably skip this part.) Go turn on a TV and look at the screen very closely. You will see a number of thin horizontal lines very closely packed together. Any picture on the TV is made up of these lines.

Inside a TV picture tube, painted on the inside of the front surface, is a substance called phosphor. Phosphor has an interesting property: when an electron hits it, the place where the electron impacted glows briefly. (A good analogy is this: A meteorite hitting the atmosphere glows briefly also; an electron alone, like a meteor without an atmosphere, doesn't glow.)

Inside the picture tube there is a device for firing electrons at the phosphor. This device is called an electron gun and sends a continual beam of electrons in a very accurate path (see Figure 1). When the electron gun fires, an electron leaves it, travels to the phosphor, and the phosphor glows briefly where it hits. Since the gun fires a steady stream of electrons, the place the gun is aimed at glows continually while the gun is firing. The picture on your TV is composed solely of these glowing dots.

At this point, we have an electron gun firing onto the phosphor of the screen. The TV picture shows one brightly glowing dot in the middle of the otherwise dark screen. If you will enter and RUN program number 1, you will get a good idea of what this looks like.

In order to draw anything on the screen bigger than a small dot, other areas of the screen must also be energized by the beam. This is done with charged "deflection plates", which bend the beam of electrons, causing the glowing dot's position to move on the screen. When the dot is moved from one point to another, a line appears; this is because the beam of electrons lights the dots in between the starting and ending points on the way. These individually lit dots appear to be a solid line because they are packed so closely together. Enter and RUN program number 2 to get a line drawn on the screen. If you have a very sharp picture, you wjll be able to see the individual dots.

However, if we trace the line just once, it will stop glowing quickly because if there aren't any electrons hitting the phosphor, it stops giving off light. In order to display a line that does not fade, the electron beam must hit the glowing dots 30 or more times a second. At that speed, or faster, the phosphor doesn't have time to fade out before the beam energizes again. If the line isn't retraced, or "refreshed", 30 or more times per second, it will visibly begin to flicker.

Any steady image you see on the TV is being continuously refreshed. The most common refresh rate is 60 times per second. If this refreshing process stops, the TV screen will quickly go blank. Television stations send information continuously to the TV, even when the screen is "frozen" (during a test pattern, for instance). An Atari continually sends signals that mean "READY" to the TV, over and over, 60 times per second when the TV displays "READY" right after you switch it on.

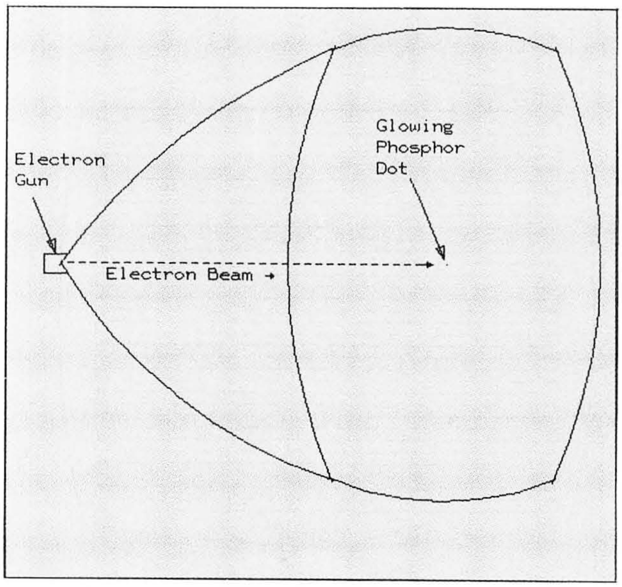


Figure 1.

```
10 GRAPHICS 8+16
20 SETCOLOR 2,0,0
30 SETCOLOR 1,0,14
40 COLOR 1
50 PLOT 160,92
60 GOTO 60
```

Program 1.

```
10 GRAPHICS 8+16
20 SETCOLOR 2,0,0
30 SETCOLOR 1,0,14
40 COLOR 1
50 PLOT 1,96
60 DRAWTO 319,96
70 GOTO 70
```

Program 2.

The electron beam is moved in a standard pattern by the deflection plates. The beam starts at the top left of the screen. It scans horizontally across the top right of the screen, and shifts down one line. It then scans from left to right again. The beam does not scan from right to left. It moves back to the left hand side before scanning again and does not just scan backwards. Then the beam traces the next line down, and continues until it reaches the end of the screen. This scanning process is called "raster scan", and the lines themselves are called "scan lines" (Figure 2).

When the beam is being traced in this fixed path, the electron gun's intensity is being varied. When many electrons hit the phosphor, it glows brightly, and when fewer electrons hit, the image is not as bright. By varying the intensity of the beam we can get shades of grey on the TV image.

In summary, the phosphor is painted on the inside of the picture tube. Electrons fired from an electron base regulate how much the phosphor glows, allowing control of brightness levels. The TV traces a beam as a vertical stack of horizontal lines, moving from left to right. The beam is then turned off momentarily to retrace to the left edge of the screen. When the beam reaches the bottom of the screen, the beam is turned off again and the electron gun starts over at the top.

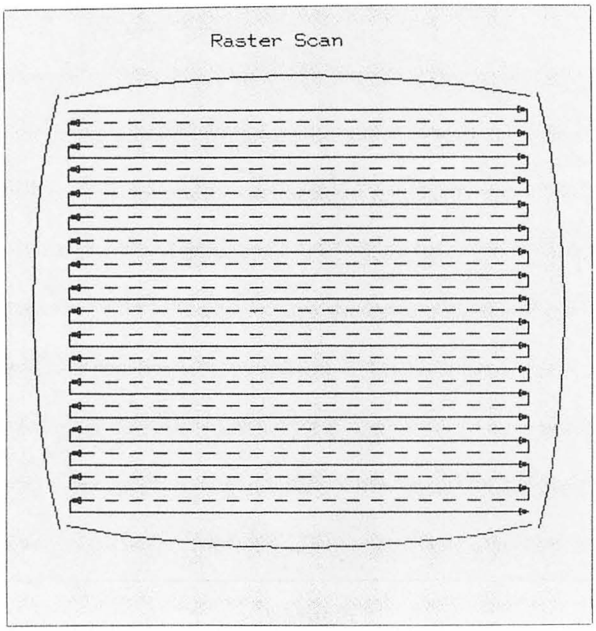


Figure 2.

Just for fun, assume that a TV screen is two feet wide and that there's 192 scan lines. Actually there are more than 192, but we will assume 192 because that is all that the Atari will allow us to use. On every refresh the electron beam traces (2 feet) x (192 scan lines), or 384 feet. Since there are 60 refreshes per second, that's 23,040 feet per second, or roughly four miles. There are 3600 seconds per hour, so a TV beam traces at 14,400 miles per hour. This is a pretty conservative estimate, too.

What about color? How does that work? Color is very similar to black and white in operation. Instead of a phosphor that only produces shades of grey, the screen is split up into many small dots. Inside each dot is a place that when hit with a beam of electrons will glow one of several colors (Figure 3). The gun is aimed very precisely at these sub-dots, so that when it's signaled, for example, to show a blue dot at a particular time in the refresh, it hits the blue "sub-point" that causes that dot to glow blue. There are a finite number of dots on the screen, because each color must be represented, packed tightly, next to each other. Each dot is in a fixed position. The Atari knows the position of all the color dots, and draws graphics or characters using them.

All a TV transmitter does is synchronize with the TV and then send it a continuing stream of color and brightness, or luminance information. The TV handles scanning back and forth and putting the information coming to it on screen at the right color/luminance. (Color and luminance information will be referred to as color/lum from now on.) The TV station doesn't specify to the TV just where a given color/lum dot should be displayed; rather, it sends that information at the time when the TV scan will have reached the proper point. The Atari works in the same manner.

An Atari needing to plot a dot at a particular point can not directly tell the TV to "put it here", and give it X and Y coordinates. Instead, it has to wait until the electron beam has reached those coordinates in its top to bottom scan, then send the TV color and luminance information for that dot. Incidentally, the Atari must immediately send information for the next dot over as well, and if the one dot is meant to stand out (as in our example) the next dot over must be dark, as set by its color and luminance information. Remember that the Atari must do all this for every lit dot on the screen sixty times per second to keep a steady TV picture.

Since the Atari conforms to TV standards, it must display everything with horizontal lines composed of individual dots. This includes lines and characters; all must be composed of dots having a certain color and certain intensity. In the next section we will examine how the Atari produces characters.

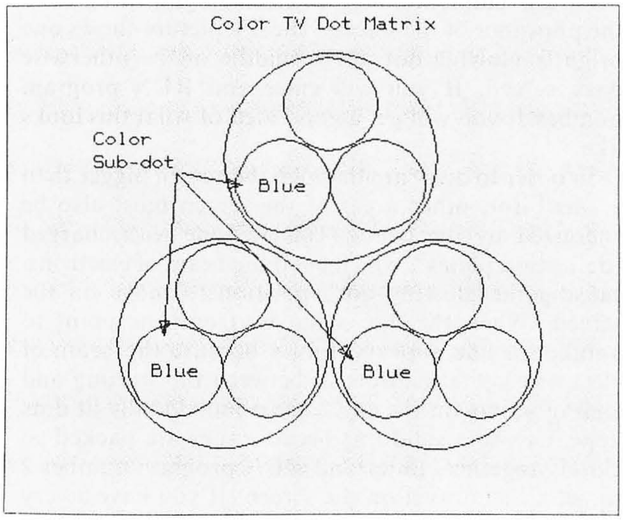


Figure 3.

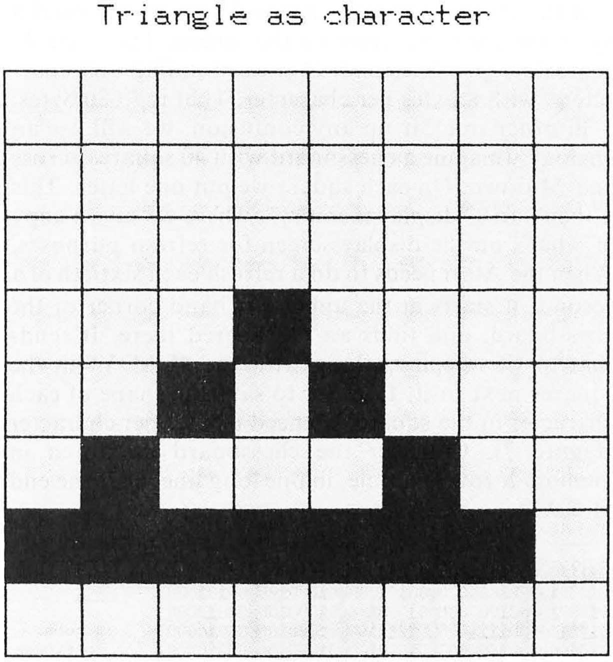
Character Generation

When you turn on your Atari, the word READY appears. We know that the letters of READY have to be made out of dots with color and luminance.

Figure 4 shows an R with a grid on top of it. Wherever in that grid the R crosses a square, the square is filled up. Thus we get a rough "R" from the shape of the squares. These squares can be thought of as dots. If we send the TV this pattern, line by line, by having it turn on the filled-in dots and leave the others off, an "R" will appear on the screen. Since we've broken the "R" up into eight horizontal segments, the "R" on the TV screen will be eight scan lines high.

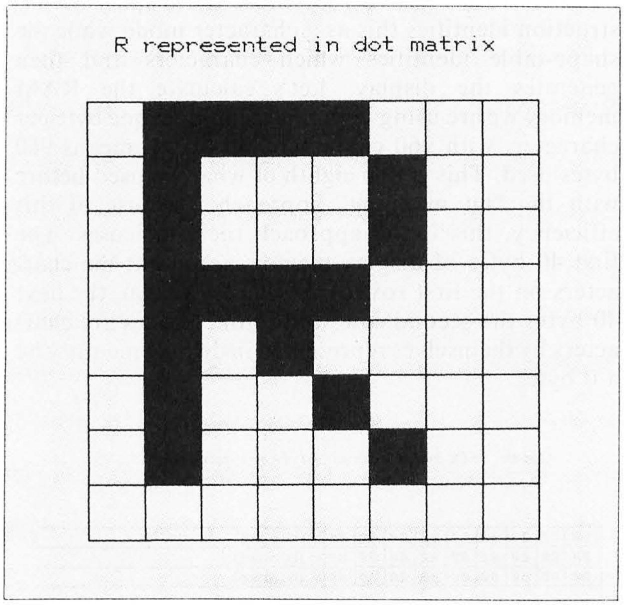
"READY" is just a bit more difficult. We must first send the top "slice" of "R", then the top slice of "E", then "A", "D", "Y", then finish out the scan line, then produce the next slice of the "R", and so forth. After eight scan lines are done, we will have our "READY" on the TV screen. We have no choice about what order the lines are plotted in (left to right, top to bottom.)

We can display any character we want, or any shape, through these methods. For example, if we wanted to display a triangle, we'd split it up into horizontal segments and send its parts as dots (Figure 5).



Triangle as character

Figure 5.



R represented in dot matrix

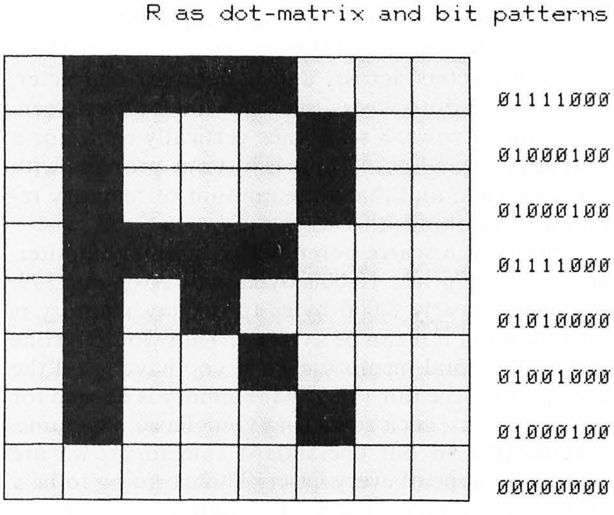
Figure 4.

Switch on your Atari and examine the READY characters. You'll be able to see the scan lines and individual dots if the picture is sharp and clear.

The Atari must refresh the TV screen 60 times each second, or the dots will stop glowing and fade away. In order to regenerate the screen, the Atari must have a copy of the information on the screen internally to send 60 times per second. The Atari saves a copy of the current TV image in memory.

Let's find out how this can be done. The Atari is trying to represent the letter "R" in memory. It thinks of an "R" as a group of eight horizontal slices of eight on/off dots. Now this corresponds nicely to our concept of bits. An unlit dot can be represented by a 0 bit, and a lit dot is represented by a 1 bit. Since there are eight bits in each slice, there is one byte (8 bits) per slice. (Do you believe in coincidences?) So let's go back to our figure of the "R" broken up into slices and represent it as bits/bytes instead (Figure 6).

It takes 8 bytes, each composed of 8 bits, to store the shape of the "R" in the Atari's memory. When refresh time comes around, the Atari takes the first byte, sends it to the display as a blank dot for each 0 bit, and a lit dot for each 1 bit. After completing the rest of the scan line, it uses the second byte, and so forth, of the "R". After 8 scan lines are done, and 8 bytes, it is finished with the "R". The "EADY" characters are also stored in memory as shapes, represented by 8 bytes each.

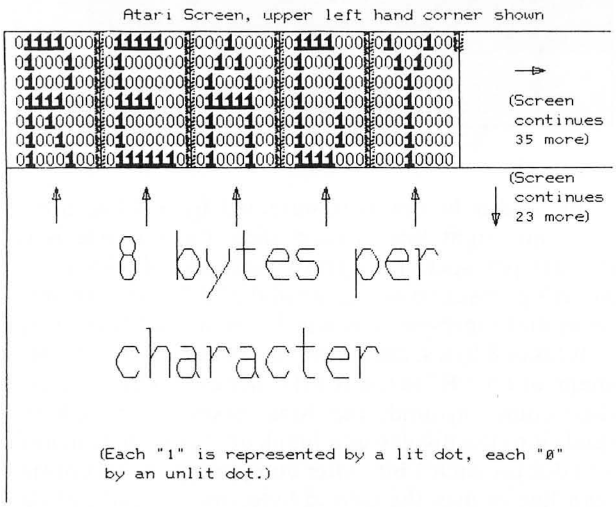


R as dot-matrix and bit patterns

Figure 6.

If the Atari used this approach, we would need 8 bytes for each character on the screen. There are 40 characters per line, and 24 lines, totaling 960 characters, with 8 bytes per character. That is 7,680 bytes.

In order to clear up any confusion, we will use an analogy. Imagine a chessboard with 40 squares across and 24 down. On each square we put one letter. This is the Atari's "display memory", where we save a copy of what's on the display screen for refresh purposes. When the Atari needs to do a refresh each sixtieth of a second, it starts at the upper left hand corner of the chessboard, and finds an "R" stored there. It sends that to the display, along with the "EADY" in the squares next to it. In order to save the shape of each character in the square, we need 8 bytes per character (Figure 7). Consider the chessboard as stored in memory a row at a time, in one long line, near the end of RAM.



Atari Screen, upper left hand corner shown

```
01111000 01111100 00010000 01111000 01000100
01000100 01000000 00101000 01000100 00101000
01000100 01000000 01000100 01000100 00010000
01111000 01111000 01111100 01000100 00010000
01010000 01000000 01000100 01000100 00010000
01001000 01000000 01000100 01000100 00010000
01000100 01111110 01000100 01111000 00010000
```

(Screen continues 35 more)

(Screen continues 23 more)

8 bytes per character

(Each "1" is represented by a lit dot, each "Ø" by an unlit dot.)

Figure 7.

This approach to display memory is called "bit mapping". The name refers to the fact that every bit in display memory corresponds to one unique dot on the screen. (The bit is "mapped" onto the screen). It gives you the capability of controlling every display dot by modifying the memory. (How many dots? Since there are 40 characters across, and 8 dots per character, there are 320 dots across one scan line of the screen. There are 24 rows, 8 scan lines vertically each, for a total of 192 scan lines.) There is just one problem with this approach, and that's the amount of memory required to do things this way.

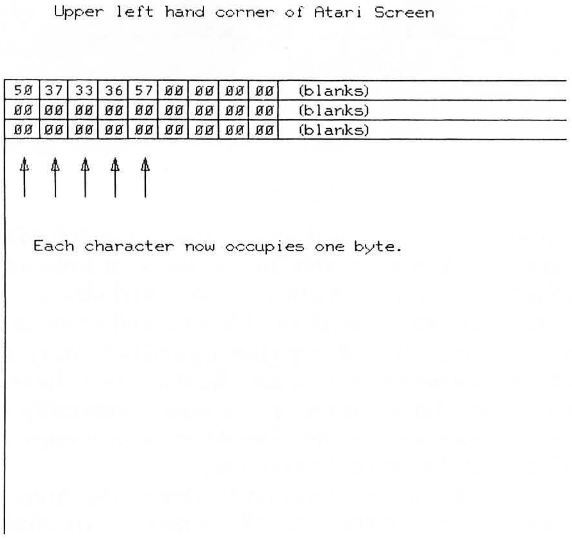
Memory is a scarce commodity on any computer. At the retail price, 16,000 bytes will cost you $99. Dedicating nearly 8,000 bytes to display memory is not very good if it can be avoided. That would be one sixth of the total memory even if you have spent the money to get the full 48K. That memory is needed for other purposes, such as storing your Basic programs.

Remember in our chessboard (memory), we are saving the shape of every letter. There's going to be a lot of redundancy. There is a number of "space" characters on the screen most of the time, each of them occupying 8 bytes. What we could do is define the shape of each character once, then in our display memory, tell the Atari where to look for the shapes of each character. The shapes of these characters do not change, so we can store them in the ROM operating system cartridge and not use the limited RAM.

Our "chessboard" used to have 8 bytes per square, saving the shape of a character. Instead of 8 bytes, let's use one. A byte can store a number from 0 to 255. For each possible character the Atari can generate, we will assign a unique number. Atari does it this way: A is 33, B is 34, and so on. (For READY: 50, 37, 33, 36, 57). Other characters, such as commas or asterisks, have their own numbers. We save those numbers in the chessboard.

The operating system ROM cartridge has a complete table of character shapes, 8 bytes per character. Given a character number stored in display memory, the Atari can determine the character's shape. From that shape the letter is displayed.

Our 40 X 24 chessboard now has numbers in it. In the upper left hand corner the first five numbers are: 50, 37, 33, 36, 57 (Figure 8). The Atari knows these numbers to be READY, because the display list instruction identifies this as a character mode while the shape-table identifies which characters and then generates the display. Let's calculate the RAM memory we are using with this technique: one byte per character, with 960 characters (40 X 24), means 960 bytes used. This is one eighth of what we used before with the "bit mapping" approach. Because of this efficiency, this is the approach the Atari uses. The first 40 bytes of display memory represent the characters on the first row of the display screen, the next 40 bytes the second row, and so on. Spaces are characters by themselves represented in display memory by a 0 byte.



Upper left hand corner of Atari Screen

5Ø	37	33	36	57	ØØ	ØØ	ØØ	ØØ	(blanks)
ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	(blanks)
ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	(blanks)

Each character now occupies one byte.

Figure 8.

Let's go through the process of displaying "READY". The Atari determines that it is time for a screen refresh. It starts with the uppermost scan line on the screen. In that line it starts with the left character's top slice. It looks to the display memory chessboard, and finds a 50 there. It looks up the 50 in its collection of character shapes, and finds the shape "R". It outputs to the display the top slice of the "R", composed of eight on-off dots from the top byte of "R". The display gets this information, composed of a group of color and luminance information, and it displays the first eight dots. The Atari sees the 37, looks it up ("E"), sends the top slice of it, and so on. It finishes the other characters, "ADY" on the first scan line, then goes on and finishes that scan line by displaying the spaces. For the second scan line on the screen, it uses the second slice of the "R", then the "EADY", and so on. After eight scan lines, it is finished with the first row of characters. It repeats this process 24 times, for there are 24 rows on the screen. This entire process is completed 60 times every second (Figure 9).

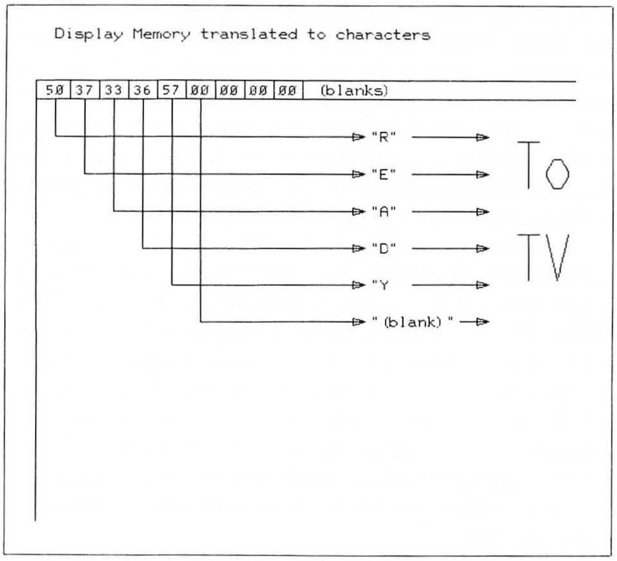

Display Memory translated to characters
5Ø 37 33 36 57 ØØ ØØ ØØ ØØ (blanks)
"R"
"E"
"A"
"D"
"Y
" (blank) "
To
TV

Figure 9.

Dot Graphics Lines

Remember our discussion of bit-mapped memory? That is how lines are produced on the Atari. We have one bit for each dot on the screen, and if that bit is "set" (1), the Atari switches on the corresponding dot on the TV. This uses a great deal of memory. The inside back cover of the Atari Basic Manual has a chart of the amount of RAM required for each graphics mode. Graphics 8 requires 7900 bytes. This compares with the value we calculated of 7,680 bytes. (The difference is due to various tables and other information the Atari must keep track of in this mode.) Also examine the number of displayable dots in X and Y directions. There are 320 horizontal dots and 192 scan lines. That's 40 characters X 8 dots per character (320), and 24 lines X 8 scan lines (192). You now should have a good idea of what happens inside the machine. Just eliminate the character squares on the chessboard, using the bytes per square technique, and you will have 320 X 192 dots, each one represented by one bit inside the Atari. The first scan line (320 dots) thus occupies 40 bytes (40 X 8 bits is 320 bits), the next scan line the next 40 bytes and so forth.

How does the Atari display a line? It turns on the dots on the screen that the line "passes over". It does this by turning on the bits in memory and letting the video refresh circuits use that memory to refresh the screen. Run program 3, and you will be able to see the individual dots light up and watch the line shift over discrete horizontal increments to draw a diagonal.

The Atari uses display memory in one of two fundamentally different ways; "character addressing" (reserving one byte per character where the number in that byte is a unqiue character number) or "bit mapping" (reserving one memory bit for each screen dot).

```
1Ø GRAPHICS 8+16
2Ø SETCOLOR 2,Ø,Ø
3Ø SETCOLOR 1,Ø,14
4Ø COLOR 1
5Ø PLOT 1,1
6Ø DRAWTO 319,1
7Ø DRAWTO 1,191
8Ø DRAWTO 1,1
9Ø GOTO 9Ø
```

Program 3.

Another computer, the Apple, has "Lo-Res" graphics (which are character addressed) and "Hi-Res" graphics, which are bit-mapped. There is no middle ground, just one or the other. Many people, familiar with the Apple, ask about the Atari's "Hi-Res" graphics. They are thinking in Apple terms, and these just do not apply as well to the Atari. The answer to the above question is "which Hi-Res mode are you referring to?"

Dot Graphics Lines

There are 14 different display operation modes in the Atari. Some are character addressed, some are bit-mapped, and some are in between. Let's restrict ourselves for now to the Basic graphics modes, which you are probably familiar with. We will return to the modes the Basic manual does not tell you about. All these different modes give you a great deal of flexibility and ease in doing complex graphics.

The character modes, 0, 1, and 2, are pretty easy to understand. Mode 0 we have been looking at. Mode 1 is just mode 0 characters stretched out to twice their width. Mode 2 is mode 0 characters stretched to twice their height and width. Since both 1 and 2 involve characters twice the width of graphics 0 characters, 8 X 2 = 16 dots wide, it will not surprise you to learn you can only fit half as many on one line (20). Modes 3-0 are graphics modes, and do not involve characters. In order to learn about them we will have to define a word, the "pixel".

A "pixel" is a group of screen dots that the Atari treats as one. They will all be the same color and will all be represented by just one memory location. If the single bit in memory that represents this group of dots is on, then the whole group is lit, and vice versa. Now, the size of a pixel is not fixed. Different graphics modes have different pixel sizes. A pixel can be as small as one dot or as large as 64 dots. Graphics 8 gives us the highest amount of control over the screen with one dot per pixel, allowing us to program every dot individually. In Graphics 3 each pixel is 8 rows of 8 dots. The size of the pixel determines the amount of graphics detail possible. Imagine a pixel to be a square of paper of varying size. A graphics 8 pixel would be the size of one TV dot. A graphics 3 pixel could be a quarter of an inch on a side. Everything going on the screen must be plotted using those squares. If you're using the larger squares, you are not going to be able to get a lot of detail. The smaller the square, the finer the detail you can draw with.

Fewer of the larger pixels can be fit on the TV at the same time, so it is reasonable to assume they will use less memory. If you will examine the graphics modes table on the back of the Atari manual, you will see this is the case. With finer detail and smaller pixels, more memory is used.

If you were drawing ten line bar graphs on the screen, you would not need fine detail. The pixels can be huge and it will not make any difference. If you are drawing a finely detailed picture, you will need small pixels and "high resolution", or fine control. In the first case, you can use a coarse detail graphics mode and not waste memory. Since the Atari has several modes, you can select the mode you need for the application and not be forced to use the highest resolution for all graphics.

The Atari manual states that only one graphics mode can be used at a time, with the option of putting a graphics 0 text window at the bottom of the screen. If you tried to mix them, and displayed data meant for graphics 0 in graphics 8, the character in the upper left hand corner, from READY, would be "R". In graphics mode 0, the "R" is represented by a 50, so the first byte of display memory would have that data in it. A 50 is a bit pattern of 0011 0010. If we switched to graphics 8, that would be the bit pattern we would have on the top scan line (blank, blank, dot, dot, blank, blank, dot, blank.) The other data in display memory would also be sent to the screen as random data. All data in display memory must be consistent with the graphics mode it was written in.

The Atari manual attempted to simplify matters for users and prevent them from having a lot of problems by announcing that graphics modes cannot be mixed. This is not so, but in mixing modes you have to know a lot about the machine and how televisions really work, and most people do not have that knowledge. The really fancy graphics capabilities, such as mixing modes, were left to Atari's own top programmers. That is what we are going to explore next.

We will set up a sample problem and solve it using mixed graphics modes. The problem is to draw a graph on the screen, titled with big letters, subtitled in small letters, with a finely detailed graphics plot and labels on the lower axis. You could do this all in the highest resolution mode, but you would be a lot older by the time you were finished. You would have to compose all the lettering out of individual dots, and that would take quite some time. Atari has taken a lot of the work out of this process and saved the programmer a lot of time.

The Dark Secrets of ANTIC and CTIA

Let's quickly review some of the TV concepts. There are 192 horizontal scan lines on the TV. Each scan line is composed of 320 dots. The lines are traced horizontally, left to right, one at a time. The top line is traced first, the bottom line last. This entire process happens 60 times per second.

The information sent to the TV is stored in the Atari memory in one of two ways, either by character addressing (where we define a letter by assigning a number to it, and use a shape table to plot it) or bit-mapping (where a bit in memory directly represents a pixel, or group of dots.) If we bit-map memory we must not display the bit-mapped data in character fashion, and vice versa, otherwise we will get random litter on the TV.

The main processor of the Atari is called a 6502. The 6502 moves data in and out of memory, and performs instructions at the rate of 1.7 million per second. (This does not mean that 1.7 Basic instructions are executed each second. In order to interpret and act on just one Basic instruction, many 6502 instructions must be executed.) Refreshing the TV screen is a job that taxes even this sort of speed. There are 320 dots on 192 scan lines, individually refreshed 60 times a second; 320 X 192 X 60 is 3.7 million dots each second. We want the 6502 to be doing things like running Basic programs, not worrying about the TV screen. So the people who designed the Atari gave the 6502 some help, in the form of a very fast, dumb slave computer called the ANTIC. ANTIC lives to do display work only, and, like the 6502, is a microprocessor.

Any computer has memory and a processor. The processor gets its data and instructions from memory, does its processing, and places data back into memory. The Atari has so many demands put on it by its graphics abilities that it comes equipped with two processors. The 6502 handles all the usual computer processing; executing Basic programs, writing data to disk, and so forth. The 6502 gives ANTIC a program, and places data in display memory. But once that's finished, the ANTIC gets the display from memory to the TV all by itself.

ANTIC feeds a chip called CTIA. CTIA is the color television interface which generates the output signal for the TV. It is not a processor, but a smart custom chip only found in the Atari. ANTIC gives the CTIA the data it needs to generate the 3.7 million dots every second. CTIA has to keep up with that rate, and has to be fed with data at that rate. ANTIC is in charge of memory control.

The 6502 and ANTIC share memory, and the computer is designed to keep them from trying to use the same memory at the same time. So while a refresh is going on, and ANTIC is frantically pulling data from memory and feeding CTIA, it turns off the 6502. When ANTIC is finished, it turns the 6502 back on. This process is called "direct memory access", or DMA, because the 6502, which normally manages memory, has nothing to do with it. If ANTIC's DMA is turned off, the video display would be lost immediately, because it would not have any data to refresh the display.

There are many books on 6502 machine language. It takes a while to learn machine language, and a lot of practice to become good at it. However, there are no books on ANTIC's machine language. Remember, ANTIC is a processor and has a program, written in its machine language. It isn't 6502 language, because ANTIC is tailored to display work. It is a display oriented language.

There are many ways you can use ANTIC's program. One of them is mixing graphics modes on the screen.

ANTIC's program is called a "display list". This display list is located in memory, just as the 6502's programs are. ANTIC generates displays by executing these display list instructions. A display list instruction is either one or three bytes long.

ANTIC single byte instructions generate displays and manage a "display block". A display block is a group of adjacent horizontal scan lines (a group of scan lines all together, with no spacing between them) which are all in the same graphics mode. They are the height of one data element in display memory.

A graphics 0 display is one row of characters. Since characters are 8 scan lines high, the display block height is 8 scan lines. A graphics 2 display block, where characters are 16 scan lines tall, is also 16 scan lines high. A graphics 3 display block, where pixels are 8 scan lines high is 8 scan lines, and a graphics 8 display block is 1 scan line high. Each data element in display memory produces some kind of graphic image on the screen, and the height of that image is the block height.

A display block is only one row of characters or one row of pixels in a graphics mode. There are 24 display blocks being shown on the screen in Graphics 0. Think of them as long, thin horizontal bars extending the full width of the screen. They are stacked on top of one another.

In graphics 8, there are 192 display blocks, all stacked on top of each other. This is because graphics 8 display blocks are only one scan line high. Display blocks do not have a fixed height. The height depends on the graphics mode one is in.

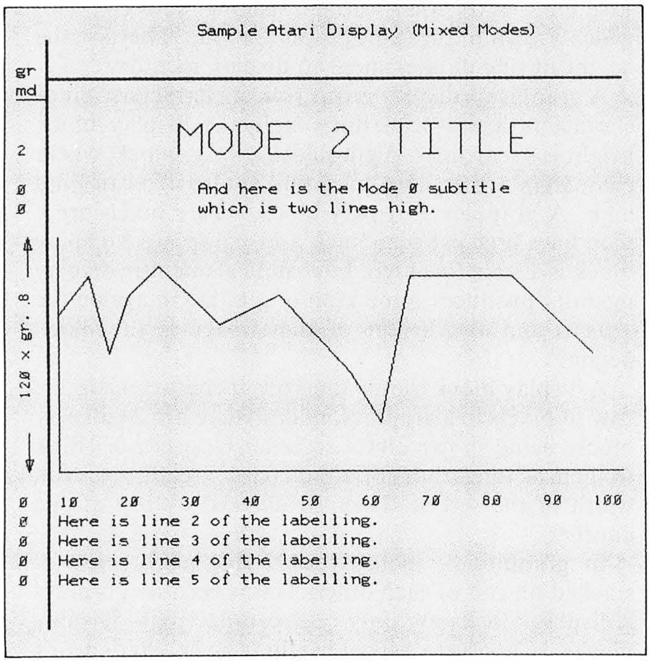
Let's look at a table of the various graphics modes from Basic and how high each mode's display block is.

Basic Graphics Mode	Size of Display Block	# Vertically Stackable
0	8 v lines/char	24 vertically stacked
1	8 v lines/char	24 vertically stacked
2	16 v lines/char	12 vertically stacked
3	8 v lines/point	24 vertically stacked
4	4 v lines/point	48 vertically stacked
5	-same-	
6	2 v lines/point	96 vertically stacked
7	-same-	
8	1 v line/point	192 vertically stacked

Inside a display block, the graphics mode cannot change. This means if something on the screen is in a given display mode, everything next to it is also in that same mode. Although the mode cannot change inside a display block, the blocks above and below that display block can be in different modes. This means we can mix modes on the screen by stacking the different display blocks that we want. This only allows us to mix modes in horizontal layers, not in vertical stripes.

Remember the graph we wanted to create? First, we would stack a graphics 2 display block (the title) on top of two graphics 0 display blocks (the subtitle) on top of a bunch of graphics 8 display blocks, on top of the graph itself (see Figure 10.) It saves a lot of time to mix graphics modes and not have to go through all the work of plotting your characters in high resolution dots.

Let's follow ANTIC through the display refresh process. We will use our earlier example, the word READY on the screen in graphics 0, in the upper left hand corner.


Sample Atari Display (Mixed Modes)

MODE 2 TITLE

And here is the Mode Ø subtitle
which is two lines high.

Here is line 2 of the labelling.
Here is line 3 of the labelling.
Here is line 4 of the labelling.
Here is line 5 of the labelling.

Figure 10.

ANTIC has an instruction in its display list which keeps it waiting until the time rolls around for a new screen refresh (every sixtieth of a second). Once this occurs, ANTIC starts work. It needs to know what to put on the very first scan line. It looks to the display list for instructions. In our example, a graphics 0 display block is first. Mode 0 has 40 characters per display block, and it takes 8 scan lines to plot the block. ANTIC turns off the 6502 and grabs 40 bytes from the start of display memory, which are the numeric representations of "READY" and 35 spaces. ANTIC knows that since this is a character mode, these 40 bytes actually represent shapes stored in a shape table, so it goes to the shape table and finds all of their shapes. Over the next eight scan lines, ANTIC plots the READY as we learned in the previous chapter. It consults the display list again for what to do with the next display block, then grabs another 40 bytes from display memory using the Graphics 0 display block, right below the first 40, and plots another line of graphics 0 characters, and continues. After 24 display blocks, the display list tells ANTIC to stop and wait for the next refresh. The 6502 is turned back on again. In graphics 0, the 6502 is turned off about 30% of the time. In graphics 8, it is turned off about 60% of the time. One way to speed up calculations on the Atari is to turn off the display completely.

If we were in graphics 8, ANTIC would have to look to the display list 192 times, and find a graphics 8 instruction that same number of times. Now if the display list tells ANTIC to first plot a mode 2 row of characters, then two mode 0 rows, then a bunch of graphics 8 blocks, it will do that. This is how we mix graphics modes.

Clearly, you have to map out your displays in advance and also map out your display memory. Since display memory is used by ANTIC as a result of plotting display blocks of data, the memory must have the exact amount of data needed stacked in the same way as the display blocks.

Go to the sample graph. A line of graphics 2 characters takes up 20 bytes of memory. That is because only 20 characters (double wide) fit on one line in graphics 2. Since this is the first display block, these bytes must be at the start of display memory, where ANTIC will start looking for data for that mode 2 line. ANTIC plots them, and while doing so, moves forward 20 bytes in memory. Next it needs 40 bytes for data for the first of two graphics 0 display blocks. That 40 bytes must immediately follow the 20 bytes of mode 2 data in memory, because that is where ANTIC will be looking for them. Next, another mode 0 line (40 more bytes), then a mode 8 line. The mode 8 line uses 40 bytes per display block (320 points, with 8 points stored per byte, is 40 bytes), bit mapped rather than character addressed. The data following the second mode 0 line must be bit mapped format, ready for graphics 0 display.

ANTIC has no idea where you want bit mapped or character addressed data to start and end other than where it is in display memory when ANTIC needs more data. You control that through the display list. If you store 41 bytes of data for a mode 0 line, it will not wrap around. ANTIC will just use that extra byte as the first data byte for the next display block and could interpret it in either graphics or character mode, depending on the display list.

Just for practice, and as an example, let's lay out a sample display list and display memory, which follows from the display list's needs, for our graph.

We design our displays around the 192 available scan lines. We will mix graphics modes, but must make the total number of scan lines used come out to exactly 192.

Here is how we allocate the 192 scan lines:

16 X 1 = 16 lines in graphics 2 for our title.
8 X 2 = 16 lines in graphics 0 for our subtitle.
120 X 1 = 120 lines in graphics 8 for the actual graph.
5 X 8 = 40 lines in graphics 0 for the labels.

This gives us a total of 192 scan lines.

There is no great penalty if we do not come out exactly at 192 scan lines. If we have a few less, the display just will not reach to the bottom of the screen. If we have a few more, we will get some bizarre displays (you may want to try this out later on). Going past 192 can result in weird things happening, as ANTIC will keep sending information after it reaches the end of the screen.

We have allocated 192 scan lines in 120 display blocks. Next we will allocate display memory. This is done by adding up the individual display block requirements.

20 bytes X 1 line = 20 bytes for the first block in graphics 2.
40 bytes X 2 lines = 80 bytes for the next two blocks in graphics 0.
40 bytes X 120 lines = 4800 bytes for the next 120 blocks in graphics 8.
40 bytes X 5 lines = 200 bytes for the last five blocks in graphics 0.

This gives us a total of 5100 bytes for display memory.

To get our graph on the screen, we set up display memory with the needed data, character addressed for modes 0 and 2 and bit mapped for the mode 8 blocks, then set up the display list with its 120 display block instructions, and finally tell ANTIC to get going. It will, and the display will pop up on the TV. A quick review of the paper and pencil process:

1. Design your display as display blocks.
2. Map out the display list from those blocks.
3. Map out display memory from the display list.

It is much easier to plot a title in large letters using graphics mode 2, where you just have to put the right 20 bytes of data into display memory and put in a mode 2 display list instruction, than to construct the letters out of individual high-resolution dots. The Atari will do all the constructing for you, and save you a lot of time. Since programmer time is becoming the most expensive factor in owing a computer, this sort of time saving is very important.

We are going to need some tables on how many bytes the various graphics modes consume. We will also have to look at how color is stored in the Atari so you will know how display memory is actually formatted, while also understanding the idea of bit mapped memory. We will then start examining and modifying display lists to get some nice effects. If you can, have an Atari available to try out the examples.

More Memory Secrets

Remember when we calculated the memory requirements of the various modes? The Atari manual chart gives numbers a little larger than the ones we calculated. For example, in graphics 0, we need only 960 bytes to store the character numbers (40 X 24 characters = 960), yet the manual said 993 bytes. The remaining bytes are the display list memory area. Let's recalculate the graphics 0 display memory and display list requirements.

A graphics 0 display list, as we will shortly see in real life, looks like this:

3 bytes which instruct ANTIC to leave the top of the screen blank.

3 bytes which instruct ANTIC where to find display memory.

24 bytes which instruct ANTIC that there are 24 graphics 0 display blocks.

3 bytes which instruct ANTIC to wait for the next refresh to begin, then go to the top of the display list and start all over again.

This totals 33 bytes.

Add these 33 bytes to the 960 bytes of the graphics 0 display memory, and you will have 993 bytes, which is what the Basic manual says. You can carry out this same calculation for other graphics modes.

Depending on the graphics mode, we have a variable number of colors available to color a given character or dot on the screen.

There are 16 colors available, numbered 0-15. There are also 8 different luminances available, numbered 0-15. Each consecutive pair of luminance values, 0 and 1, 2 and 3, and so on generate the same luminance, so there are only 8, not 16 luminances. All this color and luminance information takes 4 bits to save color information and 4 bits to save luminance information. This totals 8 bits, or one byte, to save the color and luminance for one point.

A competitive computer to the Atari stores one byte of color and luminance information for each dot in memory. We would use 320 x 192 bytes, or 61,440 bytes. Since 65,535 bytes is all of memory, we would be using nearly all of it for display! So we need to come up with a better approach.

The Atari has five "color registers" instead. These color registers are 8 bits long, and save color information in the first four bits, with luminance information in the last four. In memory when we want to specify a color, we instead specify the number of a color register that contains the color we want to have the data displayed in.

When CTIA, busily plotting data from ANTIC, looks to a graphics point and sees "01" as its color, it does not plot the point in color 1, but looks to color register 1, gets whatever color is stored in there, and plots the point in that color. A lot of memory is saved this way.

Graphics 0 and 8 do not have color information saved in their memory data. That is why it is easy to calculate their memory needs. Modes 1 and 2 actually use the top 2 bits of each character number to save a color register number. The graphics modes other than 8 work in one of two ways. They reserve either one or two bits per pixel, and use those 1 or 2 to "point to" color registers.

With 1 bit, we can specify 2 colors (0 or 1).

With 2 bits, we can specify 4 colors (00, 01, 10, 11).

Hence, if we use a 2-bit mode, we can specify a pixel to be in one of four colors, and if we use a 1 bit mode, we can specify one of two colors. In your Basic manual's graphics section, it mentions that modes 3, 5 and 7 are "four color modes", and modes 4 and 6 are "two color modes". You have just learned why. Modes 4 and 6 use less memory than their 4 color counterparts at the same resolution, for they use only 1 bit per pixel, not 2.

A typical graphics 7 (2 bits — four color) display block is 2 scan lines high and has 160 points across (2 dots per pixel horizontally and 2 scan lines per pixel vertically). Since there are 160 points, and 2 bits per point to save color information, that is 320 bits, or 40 bytes of information per block. The information is stored 4 pixels per byte, all packed in together. If we had a graphics 7 display block at the top of the screen, the first byte would contain the data for the first four points. The first point on the screen would have its color data in the first two bits of the first byte (bits 8 and 7), the second point in bits 6 and 5, and so on. If there is 00 specified as the color information, the point is not plotted. Rather, background color and luminance are used in plotting that pixel.

Below is a handy table of the various graphics modes, how they are mapped in memory, and the memory requirements.

If we store 8 points per byte, we are only using 1 bit to determine color. If we store 4 points per byte, we are using 2 bits and have a 4 color mode.

We add 9 to each display list length to handle the overhead instructions in the display list (see the previous example). These instructions are the same in each display list, hence the constant length.

The total RAM requirements will match the back of your Basic manual. The only difference will be in graphics 8, which has bytes that are unaccounted for; these are extra display list instructions made necessary by the length of display memory.

Graphics Mode	X Pts	Y Pts	Lines/ DB	# Bits/ Point	Pts/ Byte	D Mem Length	D List Length		RAM req'd (DM+DL)	RAM /1 Disp Bl
0	40	24	8	8	1	960 +	24	(+9) =	993	40
1	20	24	8	(8)	1	480 +	24	(+9) =	513	20
2	20	12	16	(8)	1	240 +	12	(+9) =	261	20
3	40	24	8	2	4	240 +	24	(+9) =	273	10
4	80	48	4	1	8	480 +	48	(+9) =	537	10
5	80	48	4	2	4	960 +	48	(+9) =	1017	20
6	160	96	2	1	8	1920 +	96	(+9) =	2025	20
7	160	96	2	2	4	3840 +	96	(+9) =	3945	40
8	320	192	1	1	8	7680 +	192	(+9) =	7891	40

Examining the Display List

The display list is written in ANTIC machine language and is a program. Do not expect the operation codes, stored in the bytes of the DL, to match what you think they should be. For example, a "0" is not a graphics 0 instruction.

ANTIC instructions come in one or three byte sizes. The three byte instructions are really one byte instructions with 2 extra bytes of data. These 2 bytes of data specify a memory location and are examined together as one 16-bit address. The address can specify any location in memory. Since ANTIC needs to be able to go anywhere in memory, these are the instructions that enable it to do so.

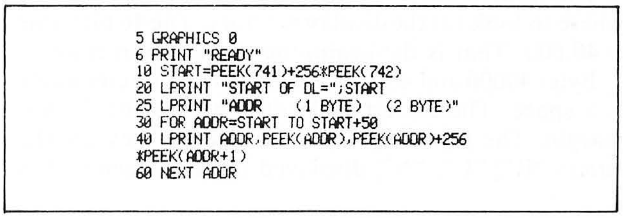
One instruction is the number "2". It is a one byte instruction that tells ANTIC to generate a graphics 0 display block, and we will be seeing a lot of them. Let's go ahead and display the display list in your machine on the printer, or on the screen if you do not have a printer. (If you do not have a printer, change every LPRINT in the program to PRINT.) Enter and run program 4.

Your printout will look like Figure 11. If you have 48K of RAM memory in your Atari, it should be identical. Let's examine the program and see how it works.

Line 10 examines two locations in memory through the PEEK statement. You may want to go back and reread the chapter on memory concepts if you are getting lost. The locations are at addresses 560 and 561, and together they specify the start of the display list. They form a 16 bit address, so line 10 takes both 8 bit values and multiplies one by 256 in order to make the number a 16 bit value for us to use. This value is the memory address of the beginning of the display list.

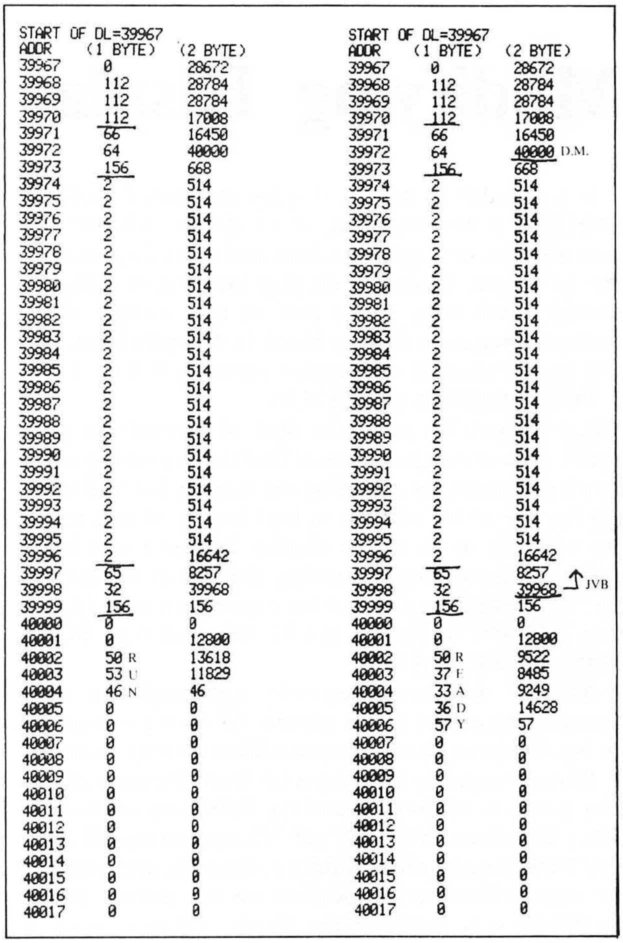
Line 10 assigns START to the address in memory where the display list begins. The program prints the address in line 20 and a title in line 30. Lines 40, 50, and 60, display the next 50 bytes' memory location, value, and 16 bit interpretation of that value, since some of these bytes will be parts of 16 bit addresses. Examining our output, we see that the byte at address 39968 contains a value of 112. The next byte is also 112, and so forth.

We have marked the two 16 bit addresses in the display list. Other than that, you can ignore the 16 bit column, as does ANTIC. Only in 3 byte instructions does it take the last two bytes and combine them to form a 16 bit address. Remember our display list is only 33 bytes long in graphics 0. We have printed out 50 locations. Let's start at the top of the display and work down, seeing what ANTIC does with each instruction. You could think of this as a program with the memory locations as line numbers. Should your Atari have less memory than mine, the memory locations will be lower, but that is no problem. Everything done here can be applied to a lower memory Atari as well.



```
5 GRAPHICS 0
6 PRINT "READY"
10 START=PEEK(741)+256*PEEK(742)
20 LPRINT "START OF DL=";START
25 LPRINT "ADDR    (1 BYTE)   (2 BYTE)"
30 FOR ADDR=START TO START+50
40 LPRINT ADDR,PEEK(ADDR),PEEK(ADDR)+256
*PEEK(ADDR+1)
60 NEXT ADDR
```

Program 4.



```
START OF DL=39967                          START OF DL=39967
ADDR    (1 BYTE)    (2 BYTE)               ADDR    (1 BYTE)    (2 BYTE)
39967    0           28672                 39967    0           28672
39968    112         28784                 39968    112         28784
39969    112         28784                 39969    112         28784
39970    112         17008                 39970    112         17008
39971    66          16450                 39971    66          16450
39972    64          40000                 39972    64          40000 D.M.
39973    156         668                   39973    156         668
39974    2           514                   39974    2           514
39975    2           514                   39975    2           514
39976    2           514                   39976    2           514
39977    2           514                   39977    2           514
39978    2           514                   39978    2           514
39979    2           514                   39979    2           514
39980    2           514                   39980    2           514
39981    2           514                   39981    2           514
39982    2           514                   39982    2           514
39983    2           514                   39983    2           514
39984    2           514                   39984    2           514
39985    2           514                   39985    2           514
39986    2           514                   39986    2           514
39987    2           514                   39987    2           514
39988    2           514                   39988    2           514
39989    2           514                   39989    2           514
39990    2           514                   39990    2           514
39991    2           514                   39991    2           514
39992    2           514                   39992    2           514
39993    2           514                   39993    2           514
39994    2           514                   39994    2           514
39995    2           514                   39995    2           514
39996    2           16642                 39996    2           16642
39997    65          8257                  39997    65          8257
39998    32          39968                 39998    32          39968 ↑ JVB
39999    156         156                   39999    156         156
40000    0           0                     40000    0           0
40001    0           12800                 40001    0           12800
40002    50 R        13618                 40002    50 R        9522
40003    53 U        11829                 40003    37 E        8485
40004    46 N        46                    40004    33 A        9249
40005    0           0                     40005    36 D        14628
40006    0           0                     40006    57 Y        57
40007    0           0                     40007    0           0
40008    0           0                     40008    0           0
40009    0           0                     40009    0           0
40010    0           0                     40010    0           0
40011    0           0                     40011    0           0
40012    0           0                     40012    0           0
40013    0           0                     40013    0           0
40014    0           0                     40014    0           0
40015    0           0                     40015    0           0
40016    0           9                     40016    0           0
40017    0           0                     40017    0           0
```

Figure 11.

The first three bytes contain the value 112. Instruction 112 calls for a display block 8 scan lines high with no characters. This tells ANTIC to take a break, do nothing with display memory, and generate 24 background color scan lines (8 scan lines for each value of 112). Many televisions "overscan", and if we do not leave a border around the display area some of that area could be lost off the screen. The 112 instructions do not look to display memory. The next 3 bytes are all one instruction. They tell ANTIC where display memory is, and tell it where to get data if it is needed. The next 24 instructions (all with the value "2") are graphics 0 display instructions. They instruct ANTIC to generate 24 graphics 0 display blocks, using data from the display memory. The last instruction in the display list is executed over and over, in a loop. This particular instruction tells ANTIC to wait until the beginning of the next screen refresh, then go to the address given in the next two bytes. The 16 bit translation of those two bytes is 39968. If you look at the top of the display list printout, you will see that this is the start of the display list.

The fourth display list instruction told ANTIC where to look for the display memory. The 16 bit value is 40,000. That is the beginning of display memory.

Bytes 40000 and 40001 are 0's. A 0 in character mode is a space. There are two leading blanks for the left margin. The next 3 bytes contain data. They are the letters "R", "U", "N", displayed on the screen and in memory when this program was executed. If you were to PEEK farther into display memory, you would find that the first 40 bytes reflect the first line displayed in graphics 0, the next 40 represent the next line, and so on. The other listing reflects what a READY would look like in display memory.

If you have less memory, for example a 32K machine, the addresses are going to be different. Your printout will tell you where the display list and memory are located. Everything is identical, except that it is located in a different part of memory.

Modifying Display Memory

It is possible to modify display memory directly by POKEing a new value in. If we do so, ANTIC will interpret the new value as data and start displaying it on the screen. Choose a display location two display blocks down (two rows) and in the middle of the screen. Since each display block is 40 bytes long, that will be the start of the display memory + 100. Type:

```
POKE 40100,46 (RETURN)
```

*Note: If you do not have 48K of memory in your Atari, do not use this. Instead find the beginning of the display memory by dumping the display list and looking for the 16 bit address in that listing. It will be the 5th and 6th bytes of the display list, and will be an address immediately following the end of the display list. For example: if the 16 bit conversion says 17,250, then add 100 to that to get 17,350, and type POKE 17350,46 (RETURN).

An "N" will have magically appeared two rows down in the center of the screen. (A 46 is the code for an N). We have just directly modified display memory.

Move around in memory a bit from the start of display memory to the end, and try POKEing in the 46 in other locations. You will get N's appearing all over. Try POKEing in other numbers than 46, and you will get other characters appearing on the screen. Do be careful to stay inside of the display memory area. If you go past display memory into the display list, unpredictable things will happen.

Try a FOR-NEXT loop from 0 to 255 and POKE the value into a display memory location. You will see all the possible letters alternating in one location. You could also use a series of memory locations with a FOR-NEXT loop to fill them with data. Here are two examples.

```
MEMLOC=XXXXX (fill in where you want data
  modified.)
FOR CHAR=0 TO 255
POKE MEMLOC,CHAR
NEXT CHAR
```

Fill a whole selection of display memory full of N's.

```
MEMSTART#XXXXX
MEMEND=MEMSTART+100
FOR LOC=MEMSTART TO MEMEND
POKE LOC,46
NEXT LOC
```

You can do similar things by POKEing into display memory in other graphics modes.

We will pick one of the 24 graphics 0 blocks, and change it to a graphics 8 block by POKEing into the display list. ANTIC will display the contents of those 40 memory locations as graphics 8 dots on one scan line (the size of graphics 8 blocks). The whole displayed area will shorten by 7 lines because the graphics 8 block is 7 lines shorter than a graphics 0 block. The letters on that line will be replaced by a graphics 8 line with dots on it, with the character data represented as dots. Above and below the graphics 8 line, there will be the usual character data. Since both graphics 0 and 8 use 40 bytes per display memory block, we do not have to worry about the start of other lines being in the wrong place.

Pick a byte in the middle of the graphics 0 instructions in the display list. We picked 39984 (see Figure 11). It can be any of the "2" graphics instructions, but if you pick one in the middle, it will show up better. Now the code for a graphics 8 display block is 15, so to modify that byte to a 15, we:

```
POKE 39984,15 (RETURN)
```

A middle line of characters is gone and there is a very small line of dots where they used to be; that's our graphics 8 display block. (If you had a blank screen in graphics 0 when you POKEd in the 15, you will not see any dots. That is because graphics 0 with a blank screen is display memory filled with 0's, and graphics 0 displays 0's as blanks.)

When you LIST a program on the screen, you will see the "black hole" effect. A line of characters will scroll up normally, hit the character line that is now a graphics 8 line, and disappear. It will be a group of dots. As the display scrolls up one more line, it will reappear out of that group of dots and something else will take its place. The dot pattern will shift also as the data on that line shifts. (Screen scrolling is accomplished through rewriting display memory). This is completely consistent and normal. Display memory has not changed, only the way ANTIC interprets that memory. If we wanted to restore the display block to graphics 0, we would type:

```
  POKE 39984,2 (RETURN)
```

to put the graphics 0 code back in.

RESET will completely rewrite the display list and clear out display memory. It is a good way to restore your display if you make a lot of mistakes.

If you change all the graphics 0 opcodes to graphics 8, your complete screen size will be 24 scan lines, or just the top one eighth of the screen, and if you LIST a program, you will get a wildly shifting dot pattern where the characters used to be. You can use a FOR NEXT loop to modify all of the display list opcodes from 2s to 8s, and then back, for a yo-yo effect.

If graphics 0 is opcode 2, and graphics 8 is opcode 15, what are 3,4,5...14? The Atari has 14 graphics modes, not the 9 that the Basic manual describes. A complete listing of graphics opcodes follows. It gives the display list opcodes, the Basic graphics number, whether it is a character or graphic mode, how high a display block it is in, the number of colors allowed, and the X and Y dimensions of the screen.

Some of these modes are just variations on other graphics modes. POKE them into the display list to try them out. One mode has 10 scan lines for letters, instead of 8. This one is for use with letters you would like to appear above or below the regular 8 scan lines, for things like exponents or subscripts.

Instead of having ANTIC show the display memory, let's have it show another part of memory. If we alter the address where ANTIC is told display memory's location, it will put whatever (probably garbage) it finds on the new location on the screen. We will choose an area of memory that is constantly changing all the time. This will make for an interesting and rapidly changing display. Type NEW for a new program, then:

```
  1 POKE 39972,1: POKE 39973,0
RUN (RETURN)
```

The reason I do these POKEs with a program is the instant either POKE is executed, the screen display will become illegible. What you will have is a rapidly flickering display reflecting low memory, where a lot of work is done. If you add:

```
  2 FOR N=1 to 65000
  3 NEXT N
```

and run it, you will watch the computer's memory as Basic executes from a neat ringside seat. For those of you with a display list in a different place, just change the appropriate locations. You should have no trouble figuring out which they should be, if you have followed the examples to this point.

If you would like to see ANTIC become misaligned with where the lines ought to start and end, try inserting a mode 2 line in the middle of the display list. Since mode 2 uses only 20 bytes per line, the remaining 20 will be picked up by the next graphics 0 line, and cause problems. Try it and watch the result.

Here are a few hints on making your own display lists and custom displays.

Start with a Basic display list longer than or equal to the length of the one you intend to have. It is very easy to shorten a display list. Just move the last instruction up a few bytes. Your display memory will be allocated by Basic this way, and you will avoid problems.

Do not try to POKE too much data. POKE is pretty slow. Until you learn machine language it is best to use PRINT or other Basic commands as much as possible.

If we were actually going to generate the graph in the example, we would start with a graphics 8 display list, move the jump instruction at the end up so we have the right number of display blocks, modify the block appropriately, then use POKEs for the titles and labels. The regular graphics 8 PLOT and DRAWTO commands would work fine for actually drawing the graph, if we modify the display slightly.

With our graphics 2 instruction at the start of the display list, we have misaligned memory with ANTIC. So move the display memory pointer back 20 bytes, and all will be well once again.

Your best bet at this point is to experiment with your own custom display lists and memory setups. The experience will be most helpful in later sections.

We will continue with further adventures in the display list. Next we will describe all the ANTIC opcodes (we have listed only the graphics related ones so far), have some discussions on how to use them, and find out some more of the tricks the display list can accomplish for us. This will help you design and implement displays faster and more effectively.

We will also discuss display list interrupts. Some spectacular display generation programs are included in this section.

Antic Code	Basic Gr. Mode	Char/Graphics	DB Lines	Colors	X	Y
2	0	Char	8	2	40	24
3	none	Char	10	2	40	odd
4	none	Char	8	4	40	24
5	none	Char	16	4	40	12
6	1	Char	8	5	40	12
7	2	Char	16	5	20	12
8	3	Graphic	8	4	40	24
9	4	Graphic	4	2	80	48
10	5	Graphic	4	4	80	48
11	6	Graphic	2	2	160	96
12	none	Graphic	1	2	160	192
13	7	Graphic	2	4	160	96
14	none	Graphic	1	4	160	192
15	8	Graphic	1	1	320	192

Display List Opcodes

There are three main groups of display list opcodes. There are also some modifiers which may be added to the basic opcodes, much like a sharp or flat may be added to a musical note. Just as certain notes may not have a sharp or flat added, certain display list opcodes may not have certain modifiers.

Here are the groups:

1. Blanking opcodes.

When ANTIC encounters one of these, it generates a certain number of blank scan lines, in the color and luminance of the background or border. It does not look to display memory or do anything else, it just generates blank scan lines. From 1 to 8 blank scan lines can be generated by these opcodes. The blank lines, like any display block, extend fully across the screen horizontally.

Modifiers: Only a display list interrupt modifier may be added to blanking opcodes.

2. Character/Graphics opcodes.

When ANTIC encounters one of these, it fetches bytes from display memory, determines the graphics mode, and puts a display on the screen. A complete list of these opcodes is available in the previous chapter.

Modifiers: Horizontal scroll, vertical scroll, load memory scan and a display list interrupt modifiers may be added to these opcodes.

3. Two special codes.

JMP is a JUMP for ANTIC. It tells ANTIC to continue looking for instructions at a different address. It is equivalent to a GOTO in the display list. It is followed by the 16 bit address of the next opcode.

JVB (Jump and wait for Vertical Blank) tells ANTIC to jump to the start of the display list, and wait for a new screen refresh to begin. It is followed by the 16 bit address of a display list to execute when the next screen refresh begins. You've seen this before, at the end of the graphics 0 display list.

Modifiers: Only a display list interrupt may be added to a jump opcode.

4. Special instructions.

JMP 01 hex (1 decimal)

JVB 41 hex (65 decimal)

Modifiers:

To add a modifier to a given opcode, just add the value given for that modifier to the base opcode, then use the total as the opcode.

1. Horizontal Scrolling.

This capability added to an instruction means that the display block may be horizontally scrolled. Add 10 hex or 16 decimal.

2. Vertical Scrolling.

This capability allows smooth vertical scrolling. Add 20 hex or 32 decimal.

3. Load Memory Scan.

(A 3 byte instruction is implied if you use this modifier.) This tells ANTIC where to find display memory, and resets ANTIC's pointer to the location, losing the current display memory pointer location. Add 40 hex or 64 decimal to the opcode.

4. Display List Interrupt.

The execution of this instruction causes ANTIC to force the 6502 to generate an interrupt. The interrupt service routine will be at the address pointed to by memory locations 200, 201 hex (512, 513 decimal).

Blank Lines

Number of blank scan lines	Hex opcode	Decimal opcode
1	00	00
2	10	16
3	20	32
4	30	48
5	40	64
6	50	80
7	60	96
8	70	112

Character / Graphics Modes

Basic Graphics Mode (if any)	Vertical Size	Horizontal Size	Colors	Graphics/ Character	Hex	Decimal
0	8	8	(2)	C	02	2
—	10	8	(2)	C	03	3
—	8	8	4	C	04	4
—	16	8	4	C	05	5
1	8	16	5	C	06	6
2	16	16	5	C	07	7
3	8	8	4	G	08	8
4	4	4	2	G	09	9
5	4	4	4	G	0A	10
6	2	2	2	G	0B	11
—	1	2	2	G	0C	12
7	2	2	4	G	0D	13
—	1	2	4	G	0E	14
8	1	1	2	G	0F	15

Special Instructions

JMP	01 hex	(01 D)
JVB	41 hex	(65 D)

We have covered "playfield graphics" (or graphics generated by the display list), ANTIC, and CTIA hardware in some depth. You now know how to generate some amazing graphics.

There is much information to present here. We will give lots of examples and ideas for their use to help you understand. The ANTIC opcodes allow you to mix graphics modes, to program display memory for mixed modes, and to format display memory.

In the next section, we will cover display list interrupts and color handling in detail as a method for achieving 128 colors on the screen at the same time. The actual goal (the 128 shades of color) is not nearly as important as the method beind it, but without the end point to work towards, the information presented is not useful or functional. By the end of the section, you will be able to generate the 128 color display and you will also have a good idea of how the Atari handles color.

After we cover display list interrupts, we will examine Player-Missile graphics. This is a separate graphics generation system that is independent of display lists and other special graphics features of the Atari. Player-Missile graphics allow high speed animation.

Notes & Discussion

1. Horizontal and vertical scrolling are good additions to graphics capabilities. They make displays easier and provide some effects that would be almost impossible to generate otherwise.

Scrolling is causing the display to appear to "roll by", so that when an object on the display comes into view, it moves across the screen and disappears on the other end. (The Atari coin-op games where you fly over enemy terrain, bombing targets that roll by underneath you, is an example of scrolling. These games could be implemented on the 400/800 using scrolling techniques.)

In order to have a display scroll, we must first send it to the screen in unmoved format, then move it, then send it again. This will cause the display to shift once. Repeatedly doing this causes a scrolling effect. All our displays, generated by ANTIC and CTIA, are stored in memory and sent to the display sixty times a second. So what we have to do is change display memory in such a way that it will cause the display on the screen to scroll.

If the display memory is changed so that all information in it is copied 40 bytes up, in graphics 0, then on the next refresh the former top line will be replaced by the information from the line below it. (Lines are 40 bytes long, remember.) You have seen this effect when the Atari scrolls something up off the screen, as happens during a long listing. If we were to move the data in the display memory up just one byte, the screen would scroll to the left, for the contents of the second byte would now be displayed in the first byte's screen position, and so on down the screen. See Figure 12.

This is a good way to do scrolling if you are working in assembly language. The amount of data that must be moved, however, (960 bytes in graphics 0) is so large that it becomes impossible for Basic to do the job quickly enough. There is a way, however, to do scrolling from Basic without moving a large block of memory. Instead of having ANTIC look at the same place in memory for display memory data and moving that data around, let's just change where ANTIC looks and leave memory alone (see Figure 13). The Atari does hot have a fixed unchangeable location in memory for display memory, unlike other machines. We can change where ANTIC looks for data with two POKEs.

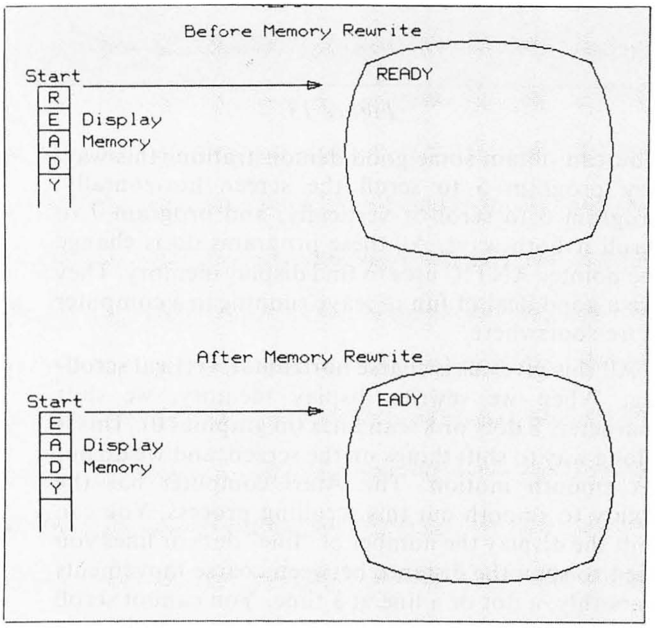


Figure 12.

For example, if we were to tell ANTIC that screen memory started one byte down from where it really did, ANTIC would skip the first real byte of screen memory, and the screen would seem to scroll to the left. ANTIC would not know the difference, yet the screen would have horizontally scrolled. If we were to tell ANTIC the screen memory starts 40 bytes down from where it really does, the screen will scroll up.

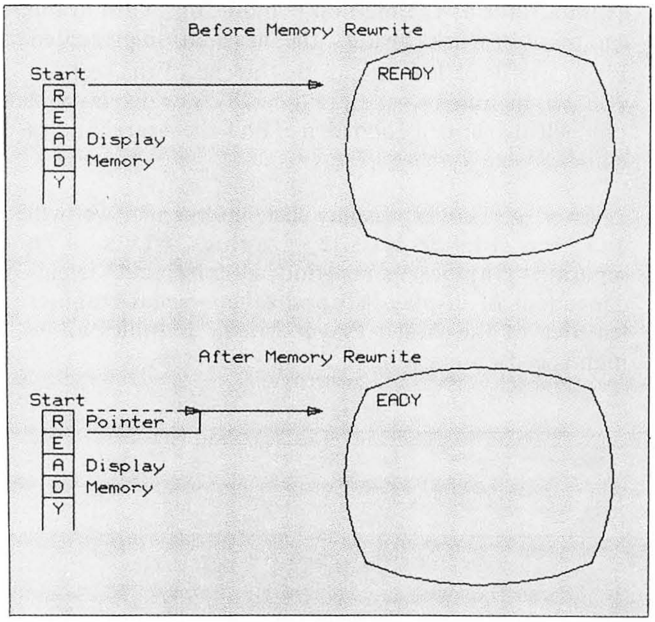


Figure 13.

You can obtain some good demonstrations this way. Try program 5 to scroll the screen horizontally, program 6 to scroll it vertically, and program 7 to scroll it both ways. All these programs do is change the pointer ANTIC uses to find display memory. They are a good deal of fun to leave running in a computer store somewhere.

All this gives us is coarse horizontal/vertical scrolling. When we rewrite display memory, we shift characters 8 dots or 8 scan lines (in graphics 0). This is a long way to shift things on the screen, and we do not get smooth motion. The Atari computer has the ability to smooth out this scrolling process. You can shift the display the number of "fine" dots or lines you need to span the distance between coarse movements smoothly, a dot or a line at a time. You cannot scroll more than the distance between one coarse scroll using the fine scroll machinery. Compare it to the fine tuning on a television set; you cannot change channels with the control, but you can smooth out the gaps between channels. Fine scrolling is limited to 0-7 dots/lines in graphics 0 or 0-16 dots/lines in graphics 2.

On the Atari scrolling is only a positive value. You cannot scroll something down using the scrolling hardware; you must start with it scrolled fully up and then scroll it "less upwards" to achieve a downward effect. How much you wish to display scrolled is written into a certain memory location.

In order to make a smoothly scrolling vertical display, we would need to select our "coarse" vertical position with the display memory and ANTIC pointer, then select how many "fine" scan lines to scroll up from that position using the scroll register. Presumably we would increment the scroll register slowly from 0 to 7, moving the display up. When we reached 7, we would rewrite display memory or change

```
20 START=PEEK(560)+256*PEEK(561)
30 REM ANTIC DISPLAY MEMORY POINTER
40 REM IS AT START+4 AND START+5
50 REM
100 X=PEEK(START+4)+256*PEEK(START+5)
110 PRINT "START OF DISP MEMORY=";X
200 FOR Y=X TO X+80
205 PRINT "POINTER=";Y
210 REM SPLIT Y UP INTO TWO BYTES
220 YHI=INT(Y/256)
230 YLO=Y-(YHI*256)
240 POKE START+4,YLO:POKE START+5,YHI
250 FOR DELAY=1 TO 20:NEXT DELAY
260 REM
270 NEXT Y
```

Program 5.

```
20 START=PEEK(560)+256*PEEK(561)
30 REM ANTIC DISPLAY MEMORY POINTER
40 REM IS AT START+4 AND START+5
50 REM
100 X=PEEK(START+4)+256*PEEK(START+5)
110 PRINT "START OF DISP MEMORY=";X
130 REM
140 REM SCROLL UP
200 FOR Y=X TO X+(40*20) STEP 40
210 GOSUB 1000
260 REM
270 NEXT Y
500 REM SCROLL DOWN
510 FOR Y=X+(40*20) TO X STEP -40
520 GOSUB 1000
530 NEXT Y
550 REM
600 REM SCROLL LEFT
610 FOR Y=X TO X+40
620 GOSUB 1000
630 NEXT Y
640 REM SCROLL RIGHT
650 FOR Y=X+40 TO X STEP -1
660 GOSUB 1000
670 NEXT Y
680 GOTO 140
990 REM CALCULATE HI, LOW BYTES
1000 YHI=INT(Y/256)
1010 YLO=Y-(YHI*256)
1030 POKE START+4,YLO:POKE START+5,YHI
1040 RETURN
```

Program 6.

```
20 START=PEEK(560)+256*PEEK(561)
30 REM ANTIC DISPLAY MEMORY POINTER
40 REM IS AT START+4 AND START+5
50 REM
100 X=PEEK(START+4)+256*PEEK(START+5)
110 PRINT "START OF DISP MEMORY=";X
130 REM
140 REM SCROLL UP
200 FOR Y=X TO X+(40*20) STEP 40
210 GOSUB 1000
260 REM
270 NEXT Y
500 REM SCROLL DOWN
510 FOR Y=X+(40*20) TO X STEP -40
520 GOSUB 1000
530 NEXT Y
540 GOTO 140
550 REM
990 REM CALCULATE HI, LOW BYTES
1000 YHI=INT(Y/256)
1010 YLO=Y-(YHI*256)
1030 POKE START+4,YLO:POKE START+5,YHI
1040 RETURN
```

Program 7.

ANTIC's pointer to get the eighth and final line. Then we would start over, incrementing our vertical scroll from 0 to 7, and continue until we were done. A downward scroll is not very different. Just move the scroll register from 7 to 0 and then rewrite memory.

The display list entry for a given display block must be modified to allow scrolling. If you write something to the scroll register, but do not change the display list, nothing happens.

Some details on how fine scrolling is implemented.

ANTIC normally displays a fixed number of scan lines per display block. For example, in graphics 0, it displays 8 scan lines. When we vertically scroll, ANTIC does not do this anymore. When ANTIC encounters the beginning of a "scrolled zone", a group of display list opcodes with vertical scroll modifiers, it treats the beginning and the end of the scrolled zone differently than it normally would to achieve the scrolling effect.

When ANTIC finds the first scroll marked display block, it does not display the normal number of scan lines for that block. It only displays the bottom "slices" of that display block, the exact number is determined by what is in the scroll register you have written to. Because the top display block is shortened, the display below that point moves up. For example, if there is a 4 in the scroll register, only scan lines 4,5,6 and 7 of the display block are shown, which are the lower slices of a character.

The display blocks in the middle of the scrolled zone (the ones with their vertical scroll modifiers set) are displayed normally, although their position is shifted up as a result of the first one having a short display block. When ANTIC reaches the end of the scrolled zone, it displays only the top few lines of the display block. If the scroll register had a 4 in it, ANTIC would only display lines 0, 1, 2, and 3; the top half of the characters. This is necessary to make the total number of scan lines in the scrolled area be the same. If the total should change, the display below the scrolled area would shift up and down with the scrolling, and it would not be limited to the zone indicated by the display list. Since the top and bottom blocks of the scroll area always have a total displayed amount of one display block, we get a fixed size scrolled zone, even though the displayed data varies. Also note that you "lose" one display block height, because the total of the two outer display blocks is now one display block height.

This is a strange way of doing things, but very effective. If you run program 8, you will see vertical scrolling in action. This program writes two separate vertical scrolled zones into the display list, then scrolls them using the register. When you run the program, you will note the size of the display shrinks by two display blocks. These are the "lost" scrolled display blocks. Note how the scrolled letters seem to disappear behind the fixed letters. They are not really disappearing, they are just not being plotted. (Figure 14).

```
5 VSCROLL=54277
10 START=PEEK(560)+256*PEEK(561)
20 REM *** PUT SOME DATA ONSCREEN.
30 GRAPHICS 0
35 FOR T=1 TO 24
36 PRINT "THIS IS LINE #";T
37 NEXT T
40 REM *** ALTER DISPLAY LIST TO V
45 REM *** IN TWO AREAS. 2 + 32 = 34
60 FOR Y=START+10 TO START+13
70 POKE Y,34
80 NEXT Y
81 FOR Y=START+17 TO START+20
82 POKE Y,34
83 NEXT Y
84 REM *** SCROLL UP
90 FOR Y=0 TO 7
100 POKE VSCROLL,Y
105 GOSUB 200
110 NEXT Y
115 REM *** SCROLL DOWN
120 FOR Y=7 TO 0 STEP -1
130 POKE VSCROLL,Y
135 GOSUB 200
140 NEXT Y
150 GOTO 90
160 REM *** SHORT DELAY LOOP
200 FOR T=1 TO 50:NEXT T
210 RETURN
```

Program 8.

Scroll Registers

Name	Address	
	Hex	**Decimal**
Vertical Scroll	**D405**	**57239**
Horizontal Scroll	**D404**	**57238**

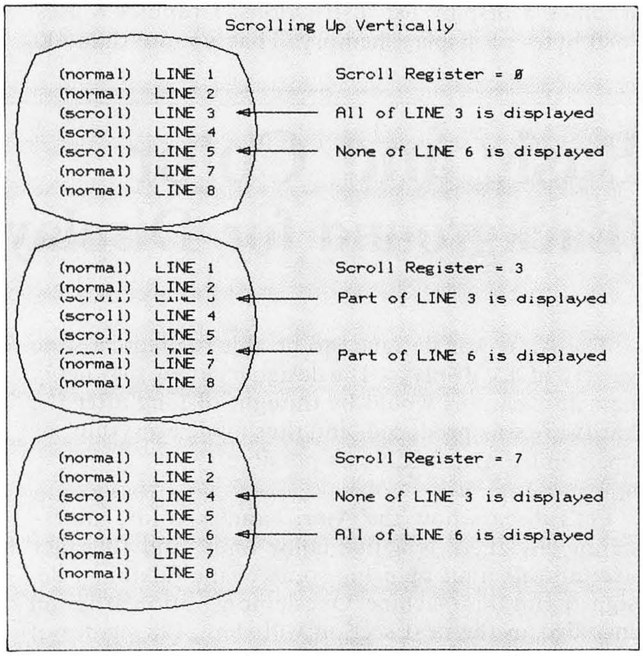


Figure 14.

2. Load Memory Scan (LMS)

This introduction is nice to know about when you are starting out and is essential later on when you start creating complex displays.

When ANTIC first learns where display memory is, it takes the 16 bit memory address and stores it in an internal (ANTIC) register. When it would like some data from display memory, perhaps 40 bytes of character data for a line of graphics mode 0, ANTIC looks to this register to find out the current line of display memory. ANTIC fetches the byte at that location. It then increments this register by one each time it gets a byte to point at a new byte.

It is tricky, because this internal register is really only 12 bits long. The other 4 bits are latches which cannot count. That means that as ANTIC goes along, if it should hit a 4K boundary in absolute memory (a 1000 hex address point),it will start all over again at the beginning of the previous 4K section of memory. This problem causes extreme frustration, even at Atari. It is quite difficult to diagnose as it resembles other problems. It is also very commonplace. Do not let your display memory cross a 4K boundary without resetting the display memory register with the load memory scan (LMS) opcode modifier. Place the 16 bit value of the next display memory in the two following bytes, low byte first, then high byte. You have seen this instruction before. It is the 66 at the beginning of the graphics 0 display list we printed. The 66 is a 2 opcode (a graphics 0 instruction), with a 64, the LMS modifier, added to it. If you actually counted the 2's in the previous example, you would find only 23 2's; the 24th display block is taken care of by the "hidden" 2 in the 66 instruction.

This instruction also accounts for some of the graphics 8 display list instructions. Graphics 8 uses 7680 bytes of display memory. That is more than 4K (4096 bytes). The graphics 8 display list must have two or more LMS instructions to reset the memory address register inside ANTIC.

The display list itself cannot cross a 1K boundary for similar reasons. ANTIC's pointer to display memory is 16 bits long, but the top 6 bits are latches only; they cannot count. You must use the JMP ANTIC instruction to pass a 1K boundary in the display list. If you start getting bizarre display list results, check this. Follow the JMP instruction with the 16 bit address to continue executing the display list.

3. Display List Interrupts

Setting this bit in an ANTIC opcode (modifiers are nothing but top bits set in the opcodes) will cause the following actions:

1. A "memory" location on the ANTIC chip is checked. If the top bit is not set, the interrupt bit is ignored.

2. If it is set, ANTIC completes the display block where it found the interrupt, up until the beginning of the last line of the block.

3. The 6502 receives a "non-maskable interrupt" (very high priority) and is sent to the memory location whose address is written into 200 and 201 hex (512,513 decimal). At the memory location whose address you POKEd into 200, 201 Hex you should have an interrupt service routine which eventually returns the 6502 to what it was doing.

Display list interrupts are incredibly powerful tools to use in your display's construction. If we needed to get a large number of colors on the screen at the same time, we could use a display list interrupt to specify a color change to occur between two display blocks. When plotting the screen, every sixtieth of a second, the Atari would change a color register (and a plotted color) in mid-refresh. Using this capability requires knowledge of a 6502 assembler language.

Basic and Color
(Background for Display List Interrupts)

The Atari was designed to be able to create a wide variety of TV displays. The designers knew that many new applications would be thought of long after the hardware was produced, and thus made everything as open ended and flexible as possible.

Let's discuss how the Atari handles color, understand why there is a five color limit, and then get around this limit by using some of the flexibility designed into this machine. One demonstration program included in the next section will show 128 shades of color on the same screen. We will give you the tools needed to generate your own custom displays with as many colors as you like, and also provide a demonstration program called "Sunset", a multicolor display that will help to slow sales of Apples at your local computer shop.

Why five colors? When the designers of the Atari worked out the details of its color handling, they decided on a technqiue which would give the user as much flexibility as possible, rather than locking him into just one methód. They had the example of the Apple, and how it handled colors, to examine and improve upon. They decided the Apple approach was not flexible enough, and came up with their own.

Inside the Atari is stored a copy of what is currently going on the TV screen. This is called "display memory". For a given point on the screen, or a group of points, some way of determining the color to be used when plotting must be stored in this memory. In the Apple the color of the point(s) is stored directly. In the Atari, the color information is stored in a "color register". When in display memory, the color of a point is specified, and a color register number is stored rather than the actual color code.

To plot a red point, one tells the display memory that this point will be plotted in the color and brightness stored in color register 1, then one puts color red at some brightness into that color register (see Figure 15).

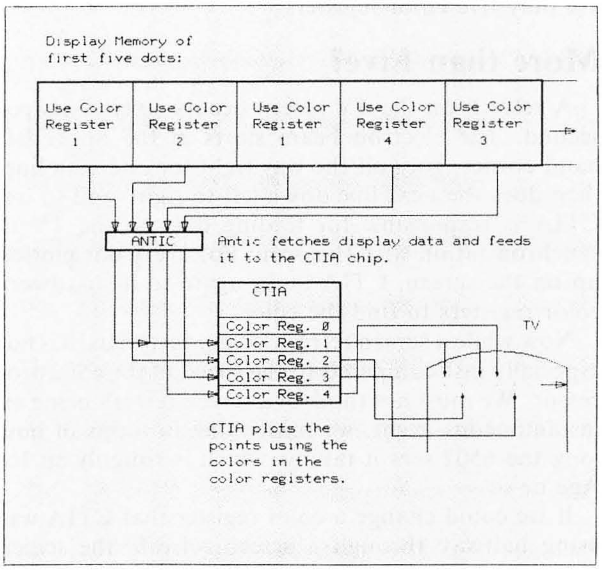


Figure 15.

There are five available color registers, numbered 0 through 4. Color register 4 is also known as the "background color register". It specifies the color and intensity for any place on the screen where nothing else is written. (In graphics modes 3 to 7, this means the color of the area between any plotted pixels. In mode 0, it means the area around the character display field, the border area, not the color behind characters.)

This approach may seem more complex than necessary, but it has advantages. It saves memory, as only two bits at most are required to specify the color of a pixel in the display memory. It adds flexibility. All we have to do to change the color of every point on the screen using the same color register is modify that register, which can be done with one POKE statement. To turn the entire screen red, then black, ("RED ALERT"), we merely need to POKE statements. In a machine without the Atari's sophistication, massive and slow rewrites of display memory would be required.

Color registers are one byte long. The upper four bits determine the color (0-15), the lower four specify the brightness. Only the top three of the four brightness bits are used, so there are eight levels of brightness, and 16 colors, or 128 total shades of color. Note that in this register, values 0-15 are color #0 in different intensities, 16-31 are color 1, and so on. If we just count this register upwards, we will pass through all 16 colors, each increasing in intensity to the highest level before moving on to the next color. If you will run program 9, you will see the screen counted through all the different shades. The table below lists the different colors.

```
20 REM ALL POSSIBLE COLORS.
25 POKE 709,14:REM SHADOW REGISTER
30 FOR C=0 TO 255 STEP 2
40 POKE 710,C:REM SHADOW REGISTER
50 PRINT "COLOR REGISTER=";C
60 FOR DELAY=1 TO 100:NEXT DELAY
70 NEXT C
```

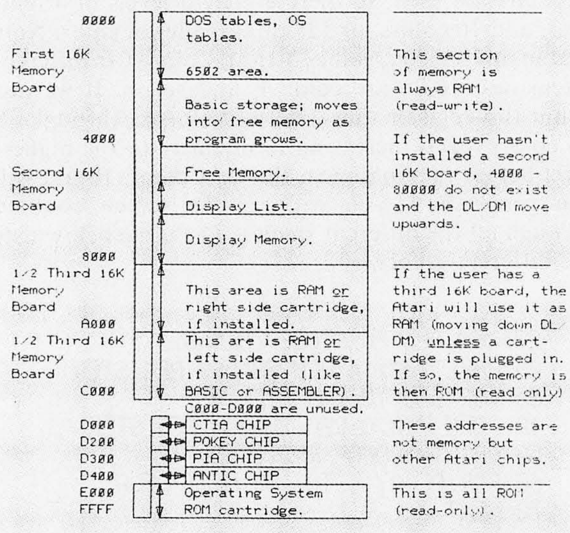
Program 9.

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

Color Register Values (as stored in upper 4 bits of color register).

How Basic Handles Colors

Up in high memory there is some memory which is not read-write, regular old RAM. When one writes or reads from these locations, one is communicating with other chips in the Atari which help support the 6502 (Figure 16). One chip, called CTIA, handles colors and graphics generation. There are five locations ("hardware color register addresses"), which are the five color registers. Now CTIA looks at these registers to find out the color needed whenever it plots a given character or point. During the refresh process of updating the screen, it looks at the registers many times, fetching the colors for displayed data.



DOS tables, OS tables.

6502 area.

Basic storage; moves into' free memory as program grows.

Free Memory.

Display List.

Display Memory.

This area is RAM or right side cartridge, if installed.

This are is RAM or left side cartridge, if installed (like BASIC or ASSEMBLER).

C000-D000 are unused.

Operating System ROM cartridge.

This section of memory is always RAM (read-write).

If the user hasn't installed a second 16K board, 4000 - 80000 do not exist and the DL/DM move upwards.

If the user has a third 16K board, the Atari will use it as RAM (moving down DL/DM) unless a cartridge is plugged in. If so, the memory is then ROM (read only)

These addresses are not memory but other Atari chips.

This is all ROM (read-only).

Atari Memory Layout and Support Chips.

Figure 16.

The operating system also maintains five "shadow registers". These are normal RAM memory locations. At the beginning of each screen refresh, these five color shadow registers are copied into the corresponding five hardware locations. Basic deals with these shadow registers.

One reason for maintaining these shadows is that the CTIA color register locations are "write only". One cannot read out of those locations where the color was just written in. They are not memory locations; they are chips, which we write to by POKEing simulated memory locations. If we wanted to read a color register and we did not have it stored somewhere (in a shadow register), we could not. Being able to read registers is handy, for example, in rotating a color from one register to the next; you use the shadow registers for this.

Basic's SETCOLOR (color reg #), (col #), (lum #) takes: (16 X color #) + lum and POKEs that value into the shadow register. One sixtieth of a second later, at the beginning of a screen refresh, the operating system copies this value into the CTIA hardware color registers and that chip then begins using it to plot data on the screen. A direct POKE to the operating system shadow locations is equivalent to a setcolor. For example: SETCOLOR 2,4,10 is the same as POKE 710,(4*16)+10.

The designers of Basic also had to come up with some way for the user to specify what color a given point or character should be. For this they have the COLOR (#) statement. It specifies which color register to use when plotting data, and remains in effect until the next COLOR statement.

The argument, or number, is not a color register number. The designers of Basic tried to keep the user away from bits and bytes discussions. The COLOR argument at first appears random.

The argument of the COLOR statement is the data that is written into display memory to specify colors. The Atari has 2 and 4 color graphics modes, using 1 or 2 bits to specify color register. For example, COLOR 0 is usually background because a 00 written into display memory plots nothing, therefore forcing the background color to appear there.

Note: SETCOLOR (n), color, lum always sets color register n. The color register number given is equivalent to the SETCOLOR register number.

In character modes (0,1,2) more than 1 or 2 bits are written into display memory. The COLOR argument is actually the character byte written in memory.

Now you understand the five color limit, for there are only five color registers.

More than Five?

A refresh on the TV screen occurs sixty times per second. The electron beam starts at the upper left hand corner, goes all the way right for one scan line, then does the next line down left to right, and so on. CTIA is responsible for feeding data to the TV in synchronization with this scan. For every dot plotted up on the screen, CTIA looks again to its hardware color registers to find the color.

Now while a screen refresh is very fast to us, it is not especially fast compared to the speed of the 6502 processor. We must not think of a screen refresh being an instantaneous event, we must think in terms of how long the 6502 sees it taking, which is roughly an Ice Age or so.

If we could change a color register that CTIA was using halfway through a screen refresh, the screen below that point would reflect CTIA using the new colors. For example, if we were in graphics 7 and modified the background hardware register halfway though a refresh from green to blue, the screen will shift from green to blue in the middle of the TV frame for all those background points (see Figure 17).

Hardware Registers (CTIA)		**O.S. Locations (Shadows)**		
D016	(53270)	2C4	(708)	PF0
D017	(53271)	2C5	(709)	PF1
D018	(53272)	2C6	(710)	PF2
D019	(53273)	2C7	(711)	PF3
D01A	(53274)	2C8	(712)	PF4(BACK)

Color Registers.

If we were to put Basic to work changing the color register as fast as it could (i.e. FOR R=0 to 255: POKE 53274,R: NEXT R) we would find that Basic would not be able to get more than one change in each frame. This is because Basic is so slow in execution, and this is why only five colors can be shown at one time if we use Basic. The five colors do not include players and missiles, which can have independent colors.

Basic needs high speed help to assist in getting a demanding job done. We have to use machine language.

Machine Language

Machine language, the human equivalent of which is called assembly language, is an art few people really love. The Atari will execute machine language instructions in times measured in the millionths of a second. Machine code is hard to understand, a pain to debug, and generally has other annoying characteristics, which is why "high level" languages such as Basic were developed in the first place.

We will provide an assembly routine that is easy to load and use from Basic. The routine will handle the demands of the 6502 so you do not have to worry about them. By setting up various tables, again from Basic, in a fairly easy way, you can have as many colors on the screen as you like, all without worrying about assembly, execution speeds, timing, and so on.

Graphics Mode	**Color Register**	**COLOR(x)**
	Character Modes	
0	0 - Unused 1 - Character lum only 2 - Char backgnd color/lum 3 - Unused 4 - Border col/lum	Not used in graphics sense.
1,2	0 - 3: Character 4 - backgnd/border	Not used in graphics sense.
	Graphics Modes	
4,6: One Bit	0: Point color/lum	COLOR 1
	1, 2, 3: Unused	
	4: Backgnd	COLOR 0
3, 5, 7: Two Bit	0: Point color/lum	COLOR 1
	1: Point color/lum	COLOR 2

COLOR # Values.

Notes:

SETCOLOR (n). color. lum always sets color register n. Hence. the color register number given is equvalent to the SETCOLOR register number.

In chapter modes (0.1.2) more than 1 or 2 bits are written into display memory. Hence the COLOR argument is actually the character byte written in memory.

Figure 17.

Display List Interrupts

The Atari computers really have two processors. One is called ANTIC and the other is the regular 6502. ANTIC has its own special language and is devoted to display work. ANTIC works with "display blocks". A display block is a group of horizontal scan lines, all in the same display mode. Think of it as a long thin horizontal bar, 8 scan lines high in graphics 0, 16 scan lines in graphics 2, and 1 scan line in graphics 8. The height is determined by the size of plotted data in the particular mode. Atari displays are composed of stacked display blocks. There are 24 stacked blocks in graphics 0, which means 24 lines of text, and 96 stacked blocks in graphics 7.

Previous sections have shown how to modify the program ANTIC uses, called a "display list", to achieve mixed graphics modes and other effects, such as scrolling. There is one change to the display list we have not yet covered, because of its complexity and the requirement of using assembly language. The remaining topic is the display list interrupt. In assembly language, a display list interrupt is given the label *DLI.*

A display list interrupt is established by setting the top bit of a display list instruction. (To Basic programmers, this means to add 128 to the instruction.) For example, a graphics 0 instruction with a DLI added is (2 + 128), or 130. Any display list instruction can have an interrupt added.

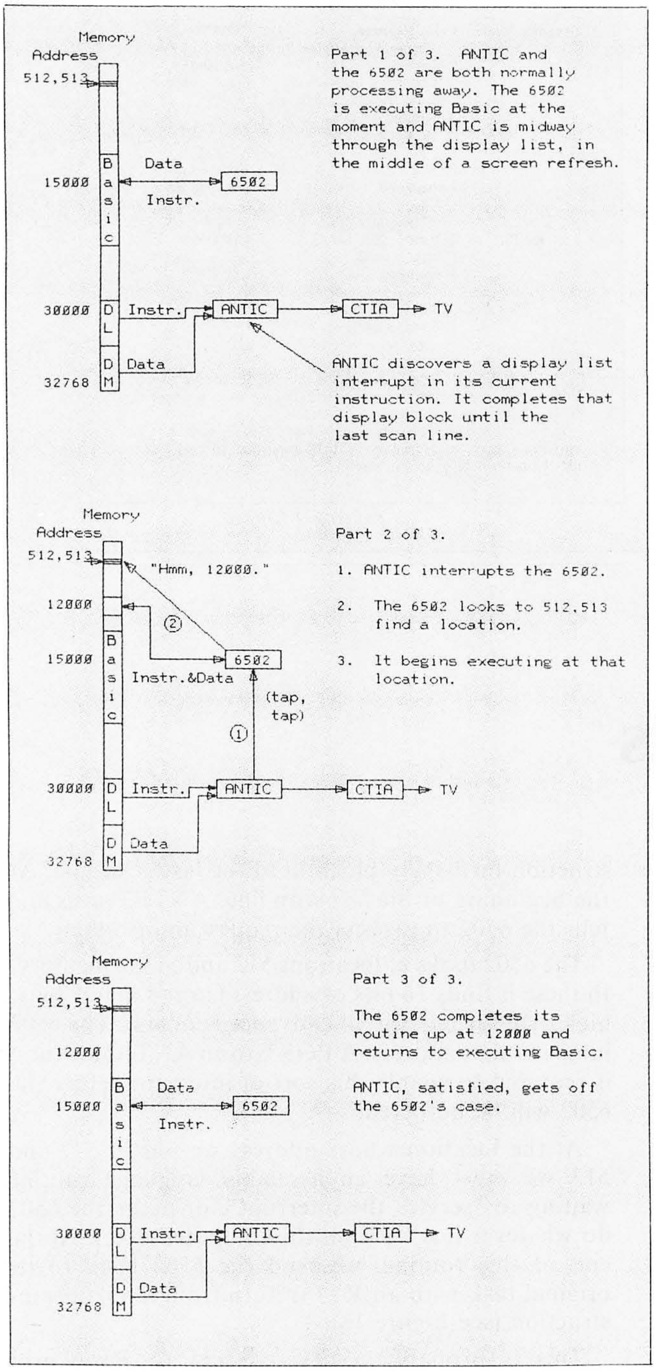
ANTIC finds the top bit of the instruction set (128 added). It goes ahead and completes the current instruction or display block until the last scan line. At the beginning of the last scan line, ANTIC turns and tells the 6502 to process the request immediately.

The 6502 looks at locations 512 and 513 in memory. In these it finds 16 bits of address (stored in low byte, high byte format, for you advanced coders). The 6502 jumps to that address. A POKE to an ANTIC location is required to enable this sort of interrupt before the 6502 will be bothered.

At the location whose address we put in 512 and 513 we must have an assembly language routine waiting to "service the interrupt", or make the 6502 do whatever it is we want the computer to do. At the end of this routine, we send the 6502 back to its original task with an RTI (return from interrupt) instruction (see Figure 18).

This is probably a new concept to Basic programmers. The best Basic analogy is the TRAP statement. TRAP specifies a line number to go to if there is an error, just as 512 and 513 specify where to go if there is an interrupt. Presumably, at that line number you have written a routine to handle errors. This is the equivalent of the interrupt service routine. You never know where an error might happen when you are executing your Basic code, just as you never know when an interrupt will occur.

The development of an assembly language routine that will be as flexible and easy to use as possible, yet run on any memory size Atari, is quite a task.



Part 1 of 3. ANTIC and the 6502 are both normally processing away. The 6502 is executing Basic at the moment and ANTIC is midway through the display list, in the middle of a screen refresh.

ANTIC discovers a display list interrupt in its current instruction. It completes that display block until the last scan line.

Part 2 of 3.

1. ANTIC interrupts the 6502.
2. The 6502 looks to 512,513 find a location.
3. It begins executing at that location.

Part 3 of 3.

The 6502 completes its routine up at 12000 and returns to executing Basic.

ANTIC, satisfied, gets off the 6502's case.

Figure 18. Display List Interrupt Processing.

Details, Details

The assembly routines must be able to fit anywhere in memory, since the memory size on different computers varies, as does program size. We've placed the entire routine into a string (PR$), and we will let the Atari decide where to put it.

A string is a collection of characters, frequently letters, numbers, and punctuation. Inside the machine all those characters are stored as bytes of memory, one per character. An assembly routine is also a collection of bytes. Since the Atari stores a string as a group of bytes, one at a time, in memory, we could make the string's characters (bytes) be the same as our assembly routine, and store the program in the string (see Figure 19).

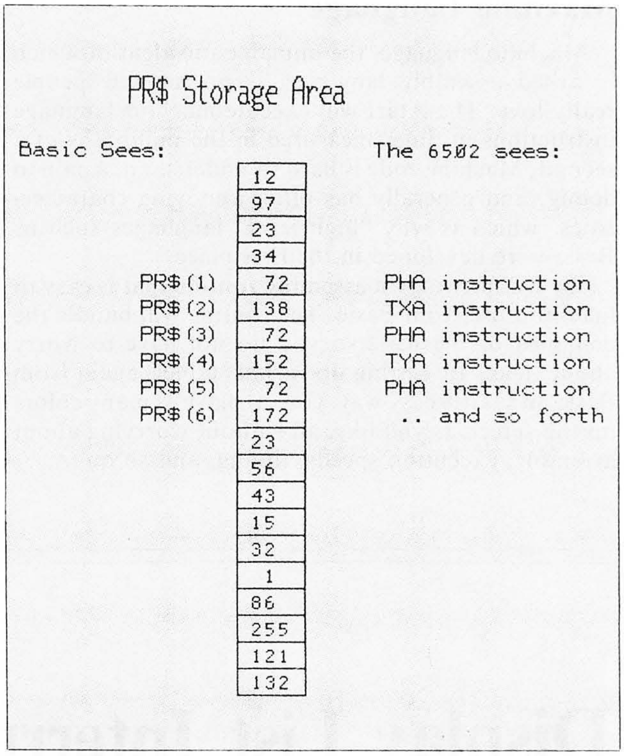
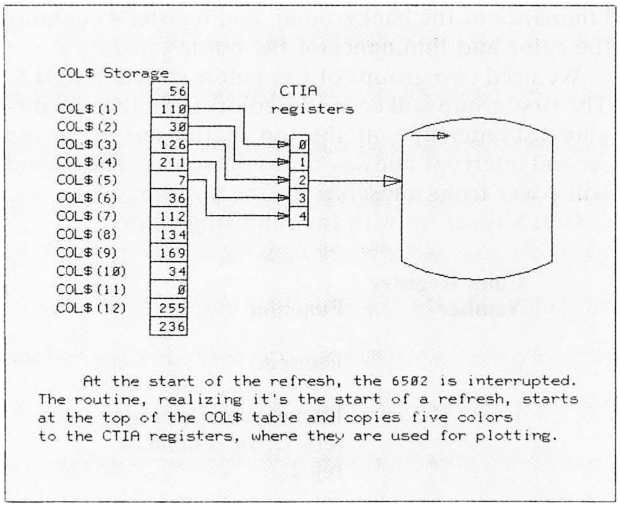

PR$ Storage Area
Basic Sees:
The 6502 sees:
12
97
23
34
PR$(1) 72 PHA instruction
PR$(2) 138 TXA instruction
PR$(3) 72 PHA instruction
PR$(4) 152 TYA instruction
PR$(5) 72 PHA instruction
PR$(6) 172 .. and so forth
23
56
43
15
32
1
86
255
121
132

Figure 19. Storing an assembly program as a string.

The Basic sub-routine to be provided reads the bytes of the assembly program into the string, one at a time. The CHR$ function takes the contents of the argument and directly stores it into the string, which is just what we want since we do not care about the actual characters. After the routine sets up the string, it links our table of colors to the program, enables display list interrupts, and returns.

The method for specifying colors is to build a table of them, five at a time (for the 5 color registers). Each time there is a display list interrupt, starting at the top of the table, the next five colors are copied into the hardware registers by the service program (in PR$). The idea is that the first display list interrupt causes the first five bytes of the table to be copied into the hardware registers, causing the colors of CTIA to change at that point. The next display list interrupt causes the second group of five color bytes to be copied into the hardware registers, again changing the colors, further down the screen. By setting interrupts and modifying the display list from Basic, you can change colors any time you like from one display block to the next. This lets you get as many colors on the screen as you want, oriented towards display blocks (see Figure 20).



At the start of the refresh, the 6502 is interrupted. The routine, realizing it's the start of a refresh, starts at the top of the COL$ table and copies five colors to the CTIA registers, where they are used for plotting.

Figure 20. COL $ Multiple Color Interrupt Handling.

How shall we set up the table? Let's use another string, COL$. The first five bytes (characters) of the string will correspond to the first display list interrupt colors, the next five bytes for the next five, and so on. Since the routine can handle a maximum of 255 bytes, this means we have a total of 255/5=51 interrupts, which is plenty.

The next problem is how the assembly routine determines when we are at the top of the screen, in order to know when to start over at the top of the table in the string. This is done by setting an interrupt at the very first display list instruction on the screen, which is a 112 (8 black scan lines). There is a hardware register called VCOUNT which tells us which scan line we are on, from the top of the screen. We read it, and if it corresponds to the blanking line at screen top, we know to start at the beginning of the table again. The routine requires this interrupt to be set. If it is not, random colors will be generated at the 6502 sails past the end of COL$, using any data in memory to determine color and luminance.

The Basic subroutine is called after you set up COL$, which is the table of colors. It requires that the location of COL$ be fixed in memory before it tells the assembly program the location of the color table. After the subroutine is executed control returns to the Basic program that called it (see Program listing 10).

Place your interrupts where you need a color register change, make sure you have the colors ready in the table (COL$), and the assembly routine will do the rest. The moment the display list is modified, the process of copying the new color codes into the hardware registers begins and the colors will change on the screen.

Basic keeps "operating system shadow registers" of the colors in the five hardware registers. When we do a SETCOLOR, we change these operating system registers. At the start of each TV refresh, sixty times a second, these shadow registers are copied by the operating system into the hardware registers. Here the colors stay, unchanged, for CTIA to use, unless something like our special assembly language routine changes them.

The routine requires the "blank 8 lines" instruction executed at the top of the frame to have an interrupt. ANTIC creates that display block, generating blank lines, and on the last scan line of the block interrupts the 6502. The 6502 looks at COL$, pulls out five bytes from the beginning of the table (since this is the top of the screen), and copies those bytes as color numbers into the hardware registers. At that point the colors

```
10000 REM ************************
10010 REM
10020 REM * DLI DRIVER / DAVE SMALL
10030 REM * YOU MUST DIM AND FILL
10040 REM * COL$ PRIOR TO GOSUB HERE
10050 REM
10060 DIM PR$(50)
10080 REM
10090 REM * READ PROGRAM INTO PR$
10100 REM
10110 READ X
10120 IF X=255 THEN 10300
10130 PR$(LEN(PR$)+1)=CHR$(X)
10140 GOTO 10110
10150 REM
10160 REM * PROGRAM AS BYTES
10170 REM
10180 DATA 72,138,72,152,72
10190 DATA 162,0,173,11,212,201,07,240,3
10200 DATA 174,01,02
10210 DATA 160,0
10220 DATA 189,03,04
10230 DATA 153,22,208
10240 DATA 232,200,192,5,208,244
10250 DATA 142,05,06
10260 DATA 104,168,104,170,104,64
10270 DATA 00,01,02,03,04,05
10280 DATA 255
10290 REM
10300 REM * LINK COL$ TO PR$
10310 REM
10320 P=ADR(PR$)
10330 PHI=INT(P/256)
10340 PLO=(P-PHI*256)
10350 REM
10360 C=ADR(COL$)
10370 CHI=INT(C/256)
10380 CLO=(C-CHI*256)
10390 REM
10400 REM * POKE IN COL$ ADDRESS
10410 REM
10420 PR$(21,21)=CHR$(CLO):REM LOWCOL
10430 PR$(22,22)=CHR$(CHI):REM HI COL
10440 REM
10450 REM * POKE IN PROGRAM LOAD ADDR
10460 REM
10470 PXHI=INT((P+41)/256)
10480 PXLO=(P+41)-(PXHI*256)
10490 PR$(16,16)=CHR$(PXLO):REM XLO
10500 PR$(17,17)=CHR$(PXHI):REM XHI
10510 PR$(33,33)=CHR$(PXLO):REM XLO
10520 PR$(34,34)=CHR$(PXHI):REM XHI
10530 REM
10540 REM * POKE IN INTERRUPT ADDRESS
10550 REM
10560 POKE 512,PLO
10570 POKE 513,PHI
10580 REM
10590 REM * ENABLE INTERRUPTS (ANTIC)
10600 REM
10610 POKE 54286,128+64
10620 REM
10630 REM * ALL SET! RETURN.
10640 REM
10650 RETURN
10660 REM **********************
```

Program 10.

```
PHA                 48H  72D
TXA                 8AH  138D
PHA                 48H  72D
TYA                 98H  152D
PHA                 48H  72D
LDX #0              A2H  162D
                    00H  0D
LDA $D40B           ADH  173D
                    0BH  11D
                    D4H  212D
CMP #7              C9H  201D
                    07H  7D
BEQ SKIP            F0H  240D
                    03H  3D
LDX $0102           AEH  174D
                    01H  1D
                    02H  2D
SKIP LDY #0         A0H  160D
                    00H  0D
LOOP LDA $0304,X    BDH  189D
                    03H  3D
                    04H  4D
STA $D016,Y         99H  153D
                    16H  22D
                    D0H  208D
INX                 E8H  232D
INY                 C8H  200D
CPY #5              C0H  192D
                    05H  5D
BNE LOOP            D0H  208D
                    F4H  244D
STX $0506           8EH  142D
                    05H  5D
                    06H  6D
PLA                 68H  104D
TAY                 A8H  168D
PLA                 68H  104D
TAX                 AAH  170D
PLA                 68H  104D
RTI                 40H  64D
(SCR1)              00H  0D
(SCR2)              01H  1D
(SCR3)              02H  2D
(SCR4)              03H  3D
(SCR5)              04H  4D
(SCR6)              05H  5D
(END)               FFH  255D
```

Program 10A. Assembler routine for display list interrupts from Program 10.

generated by CTIA change to the new values just entered.

These color values will stay on the screen until the next refresh unless we place another display list interrupt somewhere, and have five more colors ready in COL$. If we do, at the end of the display block in which the interrupt is set the color register will again change. At each refresh (every sixtieth of a second) the operating system shadow registers will once again be copied into the hardware registers, just to be replaced by our colors again, and this cycle will continue as long as the DLI instructions are in the display list.

We have to determine where on the screen we want a color change, what color registers to change, (the color registers are used differently in various graphics Modes), and then insert in the right values to make a multi-colored screen.

Our example is in graphics 0, the character mode. It will plot the top section of the screen in one color, the bottom in another.

Graphics 0 uses the five color registers as follows. Color registers 0 and 3 are unused. Register 1 determines the luminance of the characters, with same color as register 2. Register 2 contains the color and luminance of the background, and register 4 contains the color and luminance of the border.

We need two groups of five colors stored in COL$. The first group will cover the colors from the first display list interrupt, at the top of the screen, to the second interrupt midway down the screen. The second will cover from midscreen to the bottom.

COL$ is set up with the following colors:

Color Register Number	**Function**
0	Unused.
1	Luminance of characters. Color is same as reg 2.
2	Color and lum of backgnd behind characters. Not border.
3	Unused.
4	Border color and lum.

Group 1

Col reg 0, unused	COL$(1)=CHR$(0)
Luminance of characters = 10	COL$(2)=CHR$(10)
Backgnd color-lum of green, which is color #12, intensity 6.	COL$(3)=CHR$(12*16)+6
Col reg 3, unused	COL$(4)=CHR$(0)
Border Color reg, orange at 6 Orange = color 2.	COL$(5)=CHR$(2*16)+6

Group 2

Col reg 0, unused	COL$(6)=CHR$(0)
Col reg 1, lum = 10	COL$(7)=CHR$(10)
Col reg 2, red at 6	COL$(8)=CHR$((3*16)+6)
Col reg 3, unused	COL$(9)=CHR$(0)
Col reg 4, border, blue at 6	COL$(10)=CHR$((7*16)+6)

Our colors are now set up. We call the routine with GOSUB 10000. It returns control to Basic. We must now set our interrupts in the display list.

A graphics 0 display list is shown on the next page to help us visualize what we will be modifying.

We POKE (112 + 128) into START + 0, to set our first interrupt (as soon as that POKE is executed, the Atari's colors will all change to the values in the first

START=PEEK(560)+256*PEEK(561)

START	+0	112	(blank 8 lines) interrupt here required
	+1	112	
	+2	112	
	+3	66	(2, a graphics 0 instruction, + 64=lms)
	+4	data byte	
	+5	data byte	
	+6	2	graphics 0 instructions
	+7	2	
	..		
	+29	2	last graphics 0 instruction
			(halt until next refresh).

Sample Graphics Display List

five bytes, and another interrupt midway down the graphics 0 instruction at START + 6 + 15: POKE 2 + 128).

On the screen will be the colors specified, changing at the interrupt points to the new colors in COL$.

Push BREAK to halt the program. The screen stays changed! This is because the program and color strings are still in the same locations and no one told ANTIC to stop the interrupts. However, as soon as the strings get shifted or destroyed, perhaps during editing, the computer will not have an interrupt service routine anymore, and it will quietly die. (Use RESET to avoid this.) The way to exit the multicolor mode is to remove the interrupts from the display list.

For the next demonstration we will escalate things and put 16 colors on the screen at once in graphics 0. We will make each of the first 16 graphics 0 display blocks a different color.

We will need 16 interrupts; the one at the top of the screen (112), the one at the first true graphics 0 instruction, which is the 66 (66=64+2 — the 64 tells ANTIC that a display memory address follows, while the 2 is the graphics 2 instruction), and 14 more in the graphics 0 (2) instructions.

```
110 REM * 16 COLORS AT THE SAME
    TIME.
120 DIM COL$(255)
130 REM * INITIALIZE COL$ IN GROUPS
140 N=4
150 FOR C=1 TO 80 STEP 5
160 COL$(C)=CHR$(0):REM UNUSED
170 COL$(C+1)=CHR$(0):REM LUM
180 COL$(C+2)=CHR$(N):REM COLOR
190 COL$(C+3)=CHR$(0)
200 COL$(C+4)=CHR$(2):REM GREY
    BORDER
205 N=N+16
210 NEXT C
220 REM
230 REM * NOW CALL DLI HANDLER
240 REM
250 GOSUB 10000
260 REM
270 REM * NOW DO DL WORK
280 START=PEEK(560)+256*PEEK(561)
290 POKE START,112+128
295 POKE START+3,2+64+128
300 REM
310 REM * NOW POKE IN 14 MORE
    (GR/0)
320 REM
330 FOR D=START+6 TO START+6+14
340 POKE D,2+128
350 NEXT D
360 REM
370 REM PROGRAM IS NOW RUNNING.
380 REM
500 STOP
```

Program 11.

COL$ will have a length of 80 bytes, 16 interrupts multiplied by 5 colors or 80 long.

Instead of 80 sets of basic instructions, (COL$= CHR$...) we will use a short loop. Remember that adding 16 changes to the next color (see Program 11).

```
N=6 (grey color, 6 intensity)
FOR C=1 to 75 STEP 5
COL$(C)=CHR$(0) (colreg 0..unused)
COL$(C+1)=CHR$(0) (character illumination)
COL$(C+2)=CHR$(N) (color, shifts 16 each loop)
COL(C+3)=CHR$(0) (unused)
COL(C+4)=CHR$(2) (black border)
N=N+16 (bump up one color)
NEXT C
```

This loads COL$ with our desired 15 color changes, with only the background color changing between each one, at the same lum.

We call the assembly routine, then set our interrupts with another loop:

```
POKE START, 112+128 (8 blank lines, then interrupt)
POKE START+3,2+64+128 (load memory scan,
  graphics 0, and interrupt)
FOR D=START+6 to START+6+13 (14 total)
POKE D,2+128
NEXT D
```

This produces 16 colors on the screen at once. You will want to play with this routine and try different luminances for characters and background, and even the border.

One variation on this is to shift the characters through the string in a circular fashion once you have the interrupt enabled. The effect on the screen is rotating colors, with the shades of color slowly shifting up. This can be done by changing COL$, 5 bytes at a time.

(This is program 11 with a rotate).

```
COL1$=COL$(76,80) get last five characters
COL1$(6)=COL$(1,75) append first 75
COL$=COL1$ and shift it into col$
```

The rotating effect is quite spectacular and we will be using it again.

If you were mixing graphics modes you could just add an interrupt and more colors to shift displayed colors between modes. This can be quite helpful in drawing attention to a certain display. Flashing the display can be accomplished by just rewriting one byte in COL$ to 0, then back. This will change a color register on the screen immediately. All you have to do to modify the colors on the screen is modify the string.

A total of 128 shades of colors on the same screen is the limit of the Atari. The first time we saw this, the

```
110 DIM COL$(255)
120 REM * DRAW FIGURE
125 GRAPHICS 7+16
126 FOR R=0 TO 4
127 SETCOLOR R,0,R*2
128 NEXT R
129 COLOR 1
130 CL=1
140 FOR X=1 TO 120 STEP 40
150 COLOR CL
160 CL=CL+1
170 FOR Y=1 TO 95
180 PLOT X,Y
190 DRAWTO X+40,Y
200 NEXT Y
210 NEXT X
230 REM * LOAD COLORS
240 CL=0
250 FOR T=1 TO 33*5 STEP 5
260 COL$(T)=CHR$(CL):REM COL REG 0
270 COL$(T+1)=CHR$(CL+64):REM COLREG 1
280 COL$(T+2)=CHR$(CL+128):REM COLREG 2
290 COL$(T+3)=CHR$(CL+192):REM UNUSED
300 COL$(T+4)=CHR$(CL+192):REM COLREG 4
310 CL=CL+2
320 NEXT T
330 REM * CALL DRIVER
340 GOSUB 10000
350 REM * SET INTERRUPTS
360 START=PEEK(560)+256*PEEK(561)
370 POKE START,112+128
380 FOR D=START+7 TO START+7+96 STEP 3
390 POKE D,13+128:REM GR.7
400 NEXT D
410 GOTO 410
```

Program 12.

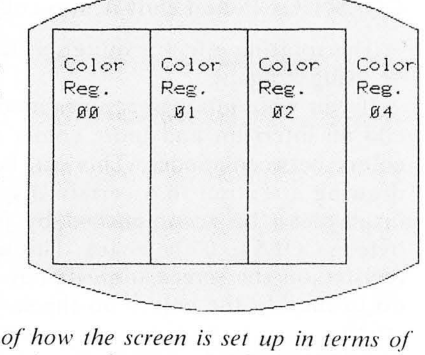
programmer dedicated the 6502 processor to updating color registers. It did nothing else. Here is our version that will run along with Basic.

Since we can only have 51 interrupts, we cannot do it the easiest way, where we shift into graphics 8 and use 128 interrupts (there are 192 scan lines in graphics 8). We use multiple color registers on the screen, side by side, and shift them four at a time, in graphics 7.

Graphics 7 uses four of the five color registers: 0, 1, 2, and 4 (background).

We generate the four blocks, each using one color register, by a simple nested loop and draw. (See program 12).

Graphics 7 has 96 display blocks, so let's set a display list interrupt every third block, for 32 total, plus one at the top of the page for 33. We will load each color register with a shade of color different from its neighbors just by counting up the 128 possible shades, and offsetting. (See program 12 listing, COL$ initialization). Remember, a change of 2 is required to change one shade of color. We poke in the DLI's as usual, this time using 13+128 (13 is for graphics 7) for 32 of them and the usual one at the top, 112+128.



In order to get 128 colors onscreen with only 32 total interrupts, we need to change 4 colors per interrupt (i.e., 32 x 4 = 128). We must also display 4 colors per line, which calls for a four color mode: graphics 3, 5, or 7.

Here's a diagram of how the screen is set up in terms of blocks of area of a given color register; this is done with Basic fill routine, but the XIO fill would work equally well.

There you have 128 shades of color at once. Let's try to rotate them.

All we need to do to rotate these colors upward is shift them by five. The value in color register four will be shifted into register 3, and as such become invisible until it is shifted into 2. A better routine could bypass the "hole" in the colors.

COL1$=COL$(6)
COL1$(161)=COL$(1,5)
COL$=COL1

This is program 12 with a rotate.

When doing string manipulations, use a scratch string and only copy it into COL$ when you're finished fiddling with it. This helps prevent the Atari from moving the string without warning with a changed length and also prevents weird screen flickering that would occur if COL$ should temporarily be too short to provide enough data.

Sunset

Let me now present the Sunset program which takes such a terrible toll on prospective Apple buyers.

It is based on a program which appeared in Creative Computing which had spirals, one inside the other. Colors were shifted between them (each spiral was in a different color register, and they were in graphics 7). The idea behind this routine is to use the shifting introduced in the previous program on the spiral routine. The initialization is a bit tricky. Each color register is started up a bit offset from the others, so that each will have a considerably different color than the others, and the background is left completely off until halfway down the screen. The colors are shifted with the top half of the string shifted up and the bottom half shifted down, an effect very much like a sunset over water. I have added a few random stars in the background on the upper half to twinkle as the color registers change. (See Program 13 on the following page).

The Atari variable table can get full of holes, if you do lots of editing, and the Atari has strange ways of cleaning up unused strings. If you start getting unexpected problems, try listing the program to storage then entering it back in with LIST and ENTER. This will clean up the variable table.

That wraps up display list interrupt concepts. You can use DLI's for other things, if you know assembly language, like switching character sets in the middle of the page.

Now that we have covered playfield graphics pretty well, let's look at redefining character sets.

```
110 REM ***
120 REM *** DAVE SMALL
130 REM ***
140 DIM TOP1$(128),BOT1$(128)
150 DIM TOP$(128),BOT$(128)
160 DIM COL$(255)
170 REM ***********************
180 REM * INITIALIZE SCREEN FOR DISP
190 GOSUB 550
200 REM * INITIALIZE COL $
210 GOSUB 790
220 REM * INITIALIZE ASSEMBLY
230 GOSUB 10000
240 REM ***********************
250 REM * DISPLAY LIST
260 ST=PEEK(560)+256*PEEK(561)
270 POKE ST,112+128:REM TOP INT(15)
280 POKE ST+3,13+64+128:REM LMS,DLI
290 FOR Y=6 TO 6+96 STEP 2:REM HI SRS
300 POKE ST+Y,13+128
310 NEXT Y
320 REM ***********************
330 REM ROTATE COL$ IN HALVES
340 FOR ROT=0 TO 300 STEP 2
350 SANDY=0
360 BOT$=COL$(1,125)
370 TOP$=COL$(126,250)
380 REM **
390 TOP1$=TOP$(1,120)
400 TOP$=TOP$(121,125)
410 TOP$(6)=TOP1$
420 TOP$(5,5)=CHR$(ROT)
426 IF ROT>255 THEN TOP$(5,5)=CHR$(0)
430 REM
440 BOT1$=BOT$(1,5)
450 BOT$=BOT$(6)
460 BOT$(121)=BOT1$
470 REM
480 COL$(1,125)=BOT$
490 COL$(126)=TOP$
500 NEXT ROT
510 GOTO 340
530 REM ************************
540 REM FROM CREATIVE COMPUTING..
550 GRAPHICS 23:DEG :SETCOLOR
     2,4,10:DIM C(3)
560 SETCOLOR 0,0,10
570 SETCOLOR 1,0,6
580 SETCOLOR 2,0,2
590 R=20:COLOR 1:C=1
600 X0=79:Y0=47
610 FOR K=0 TO 3:C(K)=K+1*2:NEXT K
620 FOR K=1 TO 3
630 X=X0+R*COS(360):Y=Y0:PLOT X,Y
640 FOR I=0 TO 5*360 STEP 75
650 X-X0+R*COS(I):Y=Y0+R*SIN(I)
660 DRAWTO X,Y
670 NEXT I:R=R+12:C=C+1:COLOR C
680 NEXT K
690 Z8=1
700 FOR LOOP=1 TO 50
710 COLOR Z8
720 X8=INT(RND(0)*159)+1
730 Y8=INT(RND(0)*47)+1
740 PLOT X8,Y8
750 Z8=Z8+1:IF Z8=4 THEN Z8=1
760 NEXT LOOP
770 RETURN
780 REM *************************
790 REM *** INIT COL$
800 FOR T=1 TO 255 STEP 5
810 COL$(T,T)=CHR$(T)
820 T1=T+80
830 T2=T+160
840 IF T1>255 THEN T1=T1-256
850 IF T2>255 THEN T2=T2-256
860 COL$(T+1,T+1)=CHR$(T1)
870 COL$(T+2,T+2)=CHR$(T2)
880 COL$(T+3,T+3)=CHR$(0)
890 COL$(T+4,T+4)=CHR$(0)
900 NEXT T
910 RETURN
```

Program 13.

Player-Missile Graphics

Introduction

Until now we have spent our time learning "playfield" graphics. Playfield graphics are graphics involving the display list and display memory. On most computers, having those two software controllable lists would mean you would have enough power to create almost any image you would ever want to. Remember that Atari is a leader in coin-operated arcade games, and most of the graphics-oriented games that people pump so many quarters into involve a lot of animation.

Animation is not an easy task on a display. One must cope with many problems to get a good animation effect. Animation involves moving objects across the display, which involves constant display memory rewrites. That by itself is not too difficult. Doing so many rewrites does tend to tie up the 6502 processor, so it is hard to get other things done at the same time. If you are animating an object in the midst of a fixed playfield image, such as a pong ball in the pong court, you have to handle such problems as how to handle an overlap between the pong ball and the playfield image.

For this, Atari added a second completely independent graphics system to the computer. This second system can be used simultaneously with the normal playfield graphics. The new graphic system is called "Player-Missile graphics".

The Atari 400/800 design was for a "video game — home computer" (to quote the hardware manual), and Atari after all is a leader in the coin-op video game field. They make the tremendously successful "Asteroids" and "Battlezone" games. In fact, the latest Atari home video system, the System X, is basically an Atari 400 under a fancy cover.

This second graphics system is intended for high speed animation and makes implementing such animation much easier.

Player-Missile Concepts

A TV picture is built up out of horizontal scan lines sent from a source synchronized with the TV's scanning. In our case, the source is the Atari, busily sending data to the TV as color/luminance information and generally trying to keep up with the tremendous amount of work to be done. ANTIC was designed to help the main 6502 processor of the Atari in generating displays. We have covered ANTIC in previous sections.

The second video chip is called CTIA. CTIA is the chip that handles assigning color and various other tasks. ANTIC is more concerned with getting data from memory in time and feeding it to CTIA. The Atari is straining to keep up with the TV. CTIA seemed to have some time left over in this process.

Atari engineers decided to give CTIA something else to do.

CTIA already keeps track of where it is horizontally on each scan line. So Atari decided to have CTIA keep track of a memory location at the same time. In this location is a number that corresponds to a horizontal position on the TV screen. Atari calls the memory location where horizontal position information is stored a "horizontal position register".

If CTIA, while busily scanning across the screen, finds that its horizontal position at the moment equals the horizontal position in this register, it will look to another register (memory location), and grab a byte of data from it. The second register is the Graphics Data Register. CTIA takes the first bit of data in this register, puts it on the screen as a dot if it is a 1, or skips it if it's a 0. For the next dot over, it does the same thing. CTIA works its way left to right on the screen, following the TV sweep, plotting 8 bits of data from the graphics data register. It plots the 8th bit first (the most significant or left hand bit). It puts these 8 dots of data on the screen at a color and luminance specified by yet another register, the "color/luminance register", which has the same format as a playfield graphics color register.

Let's say we have 150 stored in the horizontal position register, a color of bright green at 10 intensity stored in the color/luminance register, and a bit pattern of 11001011 stored in the graphics data register. When CTIA, buzzing across the screen at high speed, finds its current position is 150, it will plot dot dot (skip) dot (skip) dot dot from its current position, working to the right. The dots will be bright green. It will finish out the rest of the scan line normally. Next line down, it will also find the 150 midway across the screen, and copy the graphics data register once again onto the line. This process will happen every time it comes to 150 horizontally (see Figure 21).

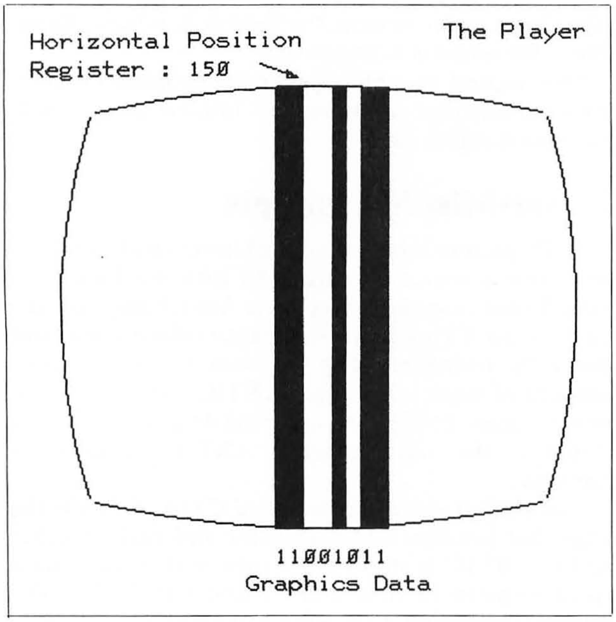


Figure 21.

The final display will be a vertical green stripe at horizontal position 150, exactly matching the bit pattern in the graphics data register. Let's go ahead and run a program to do just this, so you can see what it looks like. Enter and run program 14.

```
5 REM **** DEFINES
10 HPOS0=53248
20 BITS0=53261
30 COL0=704
40 SIZE0=53256
100 REM **** PROGRAM 1
110 REM **** GENERATE A FIXED PLAYER
120 REM
130 REM **** SET HORIZ POSITION
140 POKE HPOS0,120
150 REM **** SET COLOR
160 POKE COL0,202:REM B.GRN,INTEN=10
170 REM **** SET DATA
180 POKE BITS0,218:REM 1011 0101
190 REM **** SET SIZE
200 POKE SIZE0,0
```

Program 14.

Let's modify the program to learn about the registers it uses. Let's change the color/luminance, the graphics data, and the horizontal position registers. The effect will be, respectively, changing the color of the stripe, the bit pattern of the stripe, and the stripe's horizontal position. Run programs 15, 16, and 17 to see these effects.

```
5 REM **** DEFINES
10 HPOS0=53248
20 BITS0=53261
30 COL0=704
40 SIZE0=53256
100 REM **** PROGRAM 4
110 REM **** SHIFT PLAYER COLORS
120 REM
130 REM **** SET HORIZ POSITION
140 POKE HPOS0,150
170 REM **** SET DATA
180 POKE BITS0,203:REM 1100 1011
190 REM **** SET SIZE
200 POKE SIZE0,0
300 REM **** SHIFT PLAYER COLOR
310 FOR PCOL=0 TO 255 STEP 2
320 POKE COL0,PCOL
330 NEXT PCOL
340 GOTO 310
```

Program 15.

```
5 REM **** DEFINES
10 HPOS0=53248
20 BITS0=53261
30 COL0=704
40 SIZE0=53256
100 REM **** PROGRAM 2
110 REM **** ROTATE A PLAYER
120 REM
130 POKE HPOS0,150:REM HORIZ POS
150 REM **** SET COLOR
160 POKE COL0,202:REM B GRN,INTEN=10
190 REM **** SET SIZE
200 POKE SIZE0,0
300 REM **** MODIFY BIT PATTERN
310 FOR T=0 TO 255
320 POKE BITS0,T
325 FOR DELAY=1 TO 100:NEXT DELAY
330 NEXT T
340 GOTO 310
```

Program 16.

```
5 REM **** DEFINES
10 HPOS0=53248
20 BITS0=53261
30 COL0=704
40 SIZE0=53256
100 REM **** PROGRAM 2
110 REM **** ROTATE A PLAYER
120 REM
150 REM **** SET COLOR
160 POKE COL0,202:REM B GRN,
    INTEN=10
170 REM **** SET DATA
180 POKE BITS0,203:REM 1100 1011
190 REM **** SET SIZE
200 POKE SIZE0,0
300 REM **** ROTATE RIGHT
310 FOR T=40 TO 200
320 POKE HPOS0,T
330 NEXT T
340 GOTO 310
```

Program 17.

There is also a "size" register. The size register tells CTIA how big to make the dots when it plots them on the screen, in horizontal terms. It can make them normal size, twice normal, or four times normal. All it does is plot the same dot one, two or four times before moving on to the next one. The stripe will grow to the right, with the left position remaining constant (this is because the left border is still at position 150, where CTIA starts the display). Run program 18 to see this particular effect. A 0 in this register means x1, a 1 means x2, and a 3 means x4.

```
5 REM **** DEFINES
10 HPOS0=53248
20 BITS0=53261
30 COL0=704
40 SIZE0=53256
100 REM **** PROGRAM 5
110 REM **** MODIFY PLAYER SIZE
120 REM
130 REM **** SET HORIZ POSITION
140 POKE HPOS0,120
150 REM **** SET COLOR
160 POKE COL0,202:REM B GRN,INTEN=10
170 REM **** SET DATA
180 POKE BITS0,203:REM 1100 1011
300 REM **** SHIFT PLAYER SIZE
310 POKE SIZE0,0
315 PRINT "SIZE NORMAL"
320 GOSUB 1000
330 POKE SIZE0,1
335 PRINT "SIZE X 2"
340 GOSUB 1000
350 POKE SIZE0,3
355 PRINT "SIZE X 4"
360 GOSUB 1000
370 GOTO 310
1000 FOR Z=1 TO 1000:NEXT Z:RETURN
```

Program 18.

The stripe is known as a "player". An object only 2 bits wide, but otherwise the same as a "player," is a "missile". Hence, the name "player-missile".

What good is a green stripe in the middle of the TV that does not even shut off when we press BREAK to quit running the program? We do not have to leave data in the graphics data register on all the time. Let's say we leave it all 0's for awhile, starting from the top of the screen. CTIA will plot only the usual playfield stuff and no player dots. But if we should suddenly put a 11111111 into that register, it will plot that in scan lines from there on. We can turn it off by putting the 0 back in that register, and we have a square sitting in the middle of the screen. It will really be a stripe, with a lit square in the middle of it, but to us, it will appear to be just a square (see Figure 22).

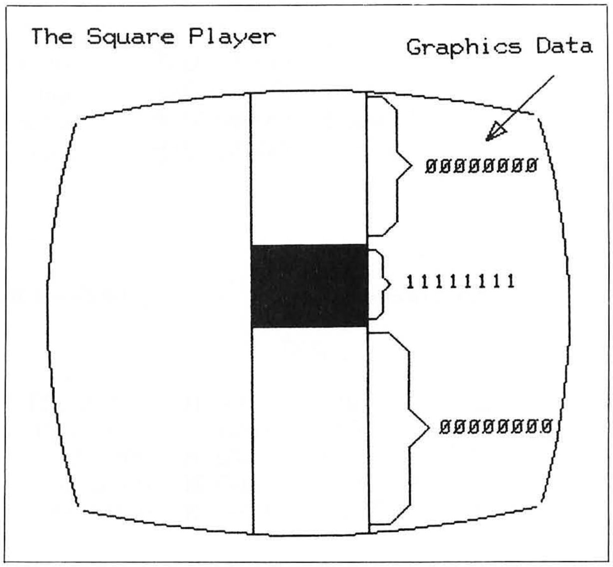


Figure 22.

We could make the shape something other than a square by putting different data in the graphics control register. We could construct a space ship out of "slices" of bits, then feed them in one at a time, one slice per scan line. This way a player could be a spaceship on the screen, which is how Star Raiders works. We could move it horizontally by changing the horizontal position register, or vertically by changing where we start putting data in. And we will have this object on our TV screen. There are 4 available players and 4 missiles, all independent. A list of the control memory locations is shown on the next page.

This table lists a number of CTIA addresses, which can also be found in later discussions of the various hardware tables. Logically, since CTIA controls players, they should be CTIA addresses.

Missiles have the same colors as their players do. Missiles can be grouped together to form a fifth 8 bit player; four missiles 2 bits wide, positioned together, equal one player 8 bits wide. All missiles have the same data, which is usually 2 bits anyway.

Some of the above demo programs will now become clear. We set player 0's horizontal position to 150 by putting that value into the approximate register with a POKE statement. The color/luminance is bright green, intensity ten, and the graphics data information is CB hex or 203 decimal. We do not write the color/luminance information to the location listed above.

Remember back when we were doing display list interrupts and we talked about operating system

Object Number	Horizontal Position Register	Graphics Data Register	Color/Lum Register	O.S.* Shadow	Size Register
Player 0	D000-(53248)	D00D-(53261)	D012-(53266)	704	D008-(53256)
Player 1	D001-(53249)	D00E-(53262)	D013-(53267)	705	D009-(53257)
Player 2	D002-(53250)	D00F-(53263)	D014-(53268)	706	D00A-(53258)
Player 3	D003-(53251)	D010-(53264)	D015-(53269)	707	D00B-(53259)
Missile 0	D004-(53252)	D011-(53265)	same as P0		D00C-(53260)
Missile 1	D005-(53253)	same	same as P1		same
Missile 2	D006-(53254)	same	same as P2		same
Missile 3	D007-(53255)	same	same as P3		same

O.S. Shadow Registers**

O.S. Location	Hardware Register		O.S. Location	Hardware Register	
–ANTIC–			–CTIA–		
22F H (559D)	D400 H	(DMA CTL)***	26F H (623D)	D01B	(PRIOR)
2F3 H (755D)	D401 H	(CHA CTL)	2C0 H (704D)	D012	(Player 0 Color)
230 H (560D)	D402 H	(Dlist L)	2C1 H (705D)	D013	(Player 1 Color)
231 H (561D)	D403 H	(Dlist H)	2C2 H (706D)	D014	(Player 2 Color)
2F4 H (756D)	D409 H	(CH BASE)	2C3 H (707D)	D015	(Player 3 Color)
			2C4 H (708D)	D016	(PlayField 0 Color)
			2C5 H (709D)	D017	(PlayField 1 Color)
			2C6 H (710D)	D018	(PlayField 2 Color)
			2C7 H (711D)	D019	(PlayField 3 Color)
			2C8 H (712D)	D01A	(PlayField 4 Backgnd Color)

Refer also to Atari columns listing hardware register addresses.

* O.S. or Operating System—A control progam that allows the computer to react with other programs, and handle input and output.

** Shadow Register—An area in memory that stores information to be transferred to a hardware control device.

*** DMA or Direct Memory Access—Allowing information in memory to be transferred from one location or device to another without using the main microprocessor. The Atari uses the Antic microprocessor to transfer information from memory to the television screen.

Abbreviations CTL = Control DList = Display List H = High Byte L = Low Byte

Player-Missile Controls

shadow addresses? These are addresses the operating system maintains in RAM that are copied into the actual hardware addresses with each screen refresh. We found that the five playfield (not player) operating system shadow registers were copied into the hardware registers on CTIA, and our display list interrupts over-wrote that data a few scan lines down. The same thing applies to player-missile colors. If they are put directly into the hardware registers, the change will last just as long as the next refresh; less than a sixtieth of a second. You will see a very brief flash of color. If you would like permanent colors for your players, use the shadow registers.

You can also use a display list interrupt routine to change a player's color halfway through a refresh. The Atari people "shadowed" a good number of registers just for the ease of having the machine restore them to the original value at the beginning of the refresh. These shadow registers include the DMA controls, the pointers to where the display list is.

The information given so far is helpful. You can get some really neat color effects running players back and forth. Try program 19 for this effect and feel free to modify it in all sorts of ways. Note the effect when two player stripes run over one another, or when a player runs over a playfield object, such as a letter or a graphics mode dot.

This is known as a "collision". When two players or missiles or playfield objects have two "on" bits tying to get through CTIA at once, we have a priority conflict that needs to be resolved. CTIA must decide whether to let the player or the letter "shine through" when a letter is plotted beneath a player. The Atari actually has incredible flexibility, and gives you many different ways to set up priorities between players, missiles,and the playfield objects. The Atari also writes data to a "collision register" to let you know that a collision occurred. You can let the Atari's hardware worry about whether there's been a collision between your spaceship and a photon torpedo (your player and your missile). You no longer have to scan tables of X and Y co-ordinates in your program to see if they have collided. Instead, just look at the collision register every now and then.

```
 20 REM DEMONSTRATES PRIORITY
 30 POKE 623,2:REM P0 HIGHEST
 40 REM POKE A 2 TO HAVE PLAYERS
    2,3
 50 REM HAVE LOWER PRIORITY THAN
    PLAYF
 60 POKE 704,4:REM COLOR P0 GREY-LO
 70 POKE 705,58:REM COLOR P1
    ORANGE-HI
 80 POKE 706,90:REM COLOR P2
    PURPLE-HI
 90 POKE 707,196:REM COLOR P3
    GREEN-LO
100 REM
110 POKE 53261,255:REM BITS P0
120 POKE 53262,255:REM BITS P1
130 POKE 53263,255:REM BITS P2
140 POKE 53264,255:REM BITS P3
150 PRINT "PLAYER 0 = GREY"
160 PRINT "PLAYER 1 = ORANGE"
170 PRINT "PLAYER 2 = PURPLE"
180 PRINT "PLAYER 3 = GREEN "
190 REM CYCLE THROUGH PRIORITIES
200 FOR X=1 TO 15
210 POKE 623,X
220 PRINT "PRIORITY CODE =";X
230 FOR T=36 TO 218 STEP 3
240 POKE 53248,20+T:REM PLAYER 0
250 POKE 53249,T:REM PLAYER 1
260 POKE 53250,218-T:REM PLAYER 2
270 POKE 53251,239-T:REM PLAYER 3
280 FOR Z=1 TO 15:NEXT Z.
290 NEXT T
300 NEXT X
310 GOTO 200
```

Program 19.

Take for example the popular game Asteroids. On the Apple a horrendous amount of time is spent checking to see if a spaceship has collided with a rock, or a missile has collided with a rock, or a player, or another ship. That is what slows the Apple game down so much; all that checking in the software takes a long time. In the Atari you do not need to. Just update the positions of the players and missiles, and check the collision registers. It will tell you all you need to know. In Star Raiders, missiles fired at the enemy are just players with high speed changes in the player data register. The enemy ships are other players, and collisions between them and missile players are checked in the hardware. That is why they run so fast.

You must select which priority scheme you would like to use with a POKE. There is a hardware location, which is located on the CTIA chip, but use the shadow register at 623 decimal instead.

Select either 8,4,2 or 1 to POKE. Adding those numbers gives wild results consisting of black regions where they overlap. (See the table below which cycles through all the different priority schemes.)

Add 10H to the number you POKE in (16 D) to let all missiles become the color of playfield 3. This way you can position them all together to be one object of the same color. Playfield 3 is not used frequently in playfield graphics, and this is why. The color is then made available for a fifth player, such as the ball in Atari Basketball.

Add 20H (32 Decimal) to have a different color (a logical OR) occur during overlap or collision.

"Play" refers to player. "P-F" refers to a playfield or display list generated, color register.

Use this priority chart after you have laid out your priorities and decided which object should be in front of another.

Location D01B (53275) contains the priority data. Use OS 632D as this address is shadowed.

Select either 8, 4, 2, or 1 to POKE with. Adding them gives odd results consisting of black overlapping regions. (Experiment!)

8	4	2	1	
P-F 0	P-F 0	Play 0	Play 0	Highest Priority
P-F 1	P-F 1	Play 1	Play 1	
Play 0	P-F 2	P-F 0	Play 2	
Play 1	P-F 3&5	P-F 1	Play 3	
Play 2	Play 0	P-F 2	P-F 0	
Play 3	Play 1	P-F 3&5	P-F 1	
P-F 2	Play 2	Play 2	P-F 2	
P-F 3&5	Play 3	Play 3	P-F 3&5	
BACKGND	BACKGND	BACKGND	BACKGND	Lowest Priority

Add 10H (16 D) to let all missiles become the color of playfield 3. Thus you can position all the misssiles for a fifth player object. Add 20H (32D) to have a different color (a logical OR) occur during an overlap.

"Play" refers to a Player. "P-F" refers to a playfield, or display list generated, object. (Remember the four available color registers?)

Use this priority chart aftr you have decided who should have priority over whom, to select the scheme that you wish.

Player-Missile DMA

So far we have vertical stripes on the screen and some interesting color effects. If Basic was not fast enough to switch color registers in the middle of a screen refresh in the display list interrupts chapter, it is unlikely it can suddenly start doing it now. So we are stuck with assembly language and tying up the 6502 turning the graphics data on and off.

You need a thorough understanding of how players are generated to use really creative graphics. The people at Atari thought of the problems they would have with tying up the 6502. Remember that we did not want to tie the 6502 up doing display work. ANTIC was created to handle the tremendous memory access needs back then. The designers of the Atari used ANTIC to help with players and missiles as well.

Remember "DMA"? That is where ANTIC took over memory in order to satisfy the needs of CTIA in doing a screen refresh. You will recall ANTIC even elbows the 6502 out of memory to get to memory more quickly. This DMA is going on all the time whenever there is a screen refresh. Now there is a memory location we can write to, DMA CONTROL (DMACTL) which controls this DMA process.

Let's say we told the ANTIC chip to completely quit using DMA. The screen would go blank. ANTIC would no longer be forcefeeding CTIA and there would be no data to plot. But the 6502 would no longer be getting shut off by ANTIC, and would not lose so much time just sitting around waiting for ANTIC to finish up. It could continue running your Basic programs, at considerably higher speed, about 30-50% faster, depending on how much graphics data would otherwise be written. This is something to keep in mind if you ever have some serious amount of processing to do in slow Basic and would like some free processing time. If you were to set up a short, custom display list and memory, or just a few lines, ending with an instruction to wait for the next refresh, that would also help free up the 6502. The less data ANTIC must fetch from display memory, the more time is available for the 6502. You could have a display list as short as one graphics instruction.

Now most articles on player-missile graphics do not cover the basics, like POKEing the hardware registers directly. They rely strictly on Player-Missile DMA. It loses a lot of the concepts involved and takes away some of the other possibilities in players. Who needs P-M DMA to set up two stripes at either ends of the screen as paddles for a "Pong" game?

Let's set up a table in memory. It will be 256 bytes long. Each byte in memory will correspond directly to one horizontal scan line on the TV. (You will note that a player extends all the way off the screen, past where ANTIC and CTIA are generating playfield.) Now if we tell them, by enabling two "switches" (actually POKEs to DMA controls) we will no longer have to transfer data directly into the CTIA graphics register. CTIA will get the graphics data byte that corresponds to the particular scan line number from ANTIC. ANTIC will look at the table, decide what scan line we are currently on, and pass the byte on the table whose number matches the scan line number to CTIA. CTIA will use that byte to plot the player or missile on the screen. Once you set up the 256 byte table and switch ANTIC on, the 6502 is once again freed up to do something else. The Player-Missile Direct Memory Access (P-M DMA) is then completely automatic (Figure 23).

Using this method, an object is defined by a few "1" bits in this table which corresponds directly to the vertical stripe on the screen. If we turn on some bits, they will show up at the next refresh. If we move them upward in the byte table, the plotted object will move up. We control the horizontal position using the horizontal position register, the color using the color register, and control the size using the size register.

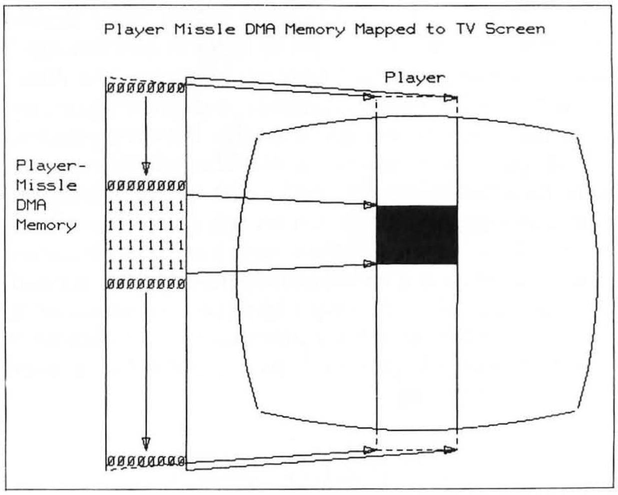
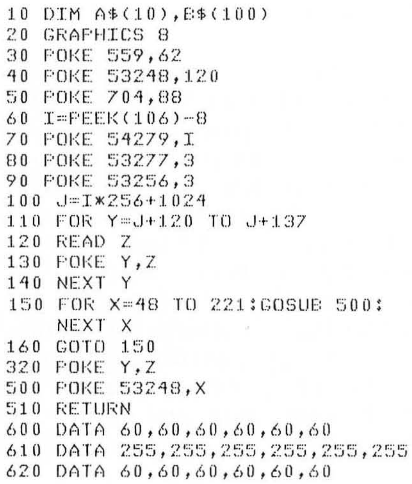


Figure 23.

Now there are a lot of things you must do to initialize this DMA process. You have to reserve a location in memory for the 256 byte tables. You have to POKE into various DMA control locations, set colors, and so on. Rather than trying to list them one by one, let's just take a working example and go over it to show how it works.

Let's look at Program 20 for a fine example of DMA use for P-M graphics.

Line 30 contains a POKE to an operating system location which instructs the system to work with a normal size playfield; i.e. 40 characters across. The location is a shadow register for a hardware register on ANTIC named DMACTL. There are two electrical switches that need to be turned on to allow P-M DMA to begin, and this is one of them. This one controls all of DMA, not just Player-Missile. The other switch is called GRACTL.



```
10 DIM A$(10),B$(100)
20 GRAPHICS 8
30 POKE 559,62
40 POKE 53248,120
50 POKE 704,88
60 I=PEEK(106)-8
70 POKE 54279,I
80 POKE 53277,3
90 POKE 53256,3
100 J=I*256+1024
110 FOR Y=J+120 TO J+137
120 READ Z
130 POKE Y,Z
140 NEXT Y
150 FOR X=48 TO 221:GOSUB 500:
    NEXT X
160 GOTO 150
320 POKE Y,Z
500 POKE 53248,X
510 RETURN
600 DATA 60,60,60,60,60,60
610 DATA 255,255,255,255,255,255
620 DATA 60,60,60,60,60,60
```

Program 20.

Line 40 is a POKE to the horizontal position register for Player 0, putting the player at 120. Line 50 is a POKE Player 0's shadow color register for a pink color. We need 256 bytes for our DMA table for Player 0. We must run all the players and missiles with DMA, so we need considerably more than 256 bytes. The total comes out to 2048 bytes for our "bitmaps" of the players.

We need to find a place in free memory to put this 2048 bytes. Remember when we were using alternate character sets that required 1024 bytes? We modified the top of the memory pointer, located in location 106, and moved it back to make sure we had an area of memory the Atari would not use. We will do the same thing here, moving the pointer back 8*256 or 2048 bytes back.

We then POKE the memory page number (address/256) into a location called PMBASE. This tells ANTIC where to start fetching data for CTIA.

We POKE a location called GRACTL, for GRAphics ConTroL register. GRACTL is the second of two switches that has to be turned on to enable P-M graphics. We POKE a 3 into there to tell ANTIC and CTIA to start using P-M DMA. We POKE player 0's size register at 53256 with a 3 to make our player 4 times normal size.

Our Player is now being plotted on the screen. Whatever junk is in memory at this point is now busily being pulled out of memory by ANTIC and fed to CTIA. The area of memory we are using is empty. The Graphics instruction at the beginning of the program, before we moved the memory pointer back, cleared it out for us. We should put some sort of bit pattern into memory to create a display. The program does that next.

Line 280 sets J equal to I (which is the start of the P-M table) * 256 (because that value was in 256 byte pages.) J points to the beginning of the P-M table in memory. We add 1024 to it, the Player 0 data is 1024 bytes from the beginning of the table, with the other players and missiles around it. The table looks like the example shown to the right.

There is also a P-M DMA mode where only 1024 bytes are used, 128 per player. In this mode each byte represents not one but two scan lines. This is known as a "double line resolution" in the manuals. Just adjust the addresses above for half as much data.

J now equals the beginning of the Player 0 bit map data. The FOR loop in the program runs from J+120 to J+137, for 18 lines in the middle of the screen. Data is read in and copied into those bytes. This data is in the form of a "Cross", where the 60's on top and bottom are: 00111100 and the 255's are: 11111111. This data will be sent to the screen in the form of dots for 1 bits and blanks for 0 bits, so at this point we have a cross on the screen, still in pink. We have a loop which pokes the horizontal position register from 48 to 221, and starts over at 48 again. This will move our player stripe, with the cross in the middle of it, across the screen (see Figure 24).

There should not be a lot of mystery left in players and missiles now. ANTIC is just POKEing our CTIA graphics data registers for us. This helps free up the 6502. We move the player horizontally with a POKE, vertically by copying bits in this table up and down, and select color and size with POKEs. We will learn how to read collisions later.

PMBASE * 256 = start of this area

BASE ADDRESS

+ 0
-- wasted space.
+ 767

+ 768
-- Missile Data. Missile data is packed side by side, 4 missiles of 2 bits per byte.

M3 : M2 : M1 : M0

+ 1023

+ 1024
-- Player 0 bitmap. Top byte is top line of TV (above viewing area).
+ 1279

+ 1280
-- Player 1 bitmap.
+ 1535

+ 1536
-- Player 2 bitmap.
+ 1791

+ 1792
-- Player 3 bitmap.
+ 2047

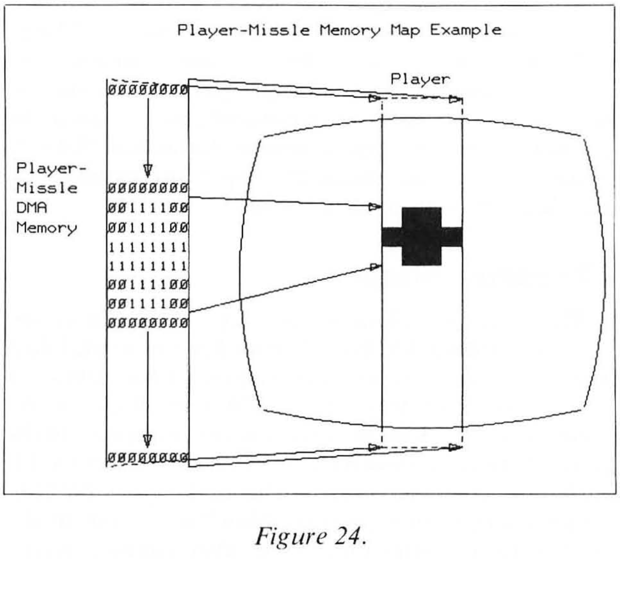


Figure 24.

The Beginner's Guide to Character Sets

An important part of the design of the Atari computers was to obtain varied and interesting graphics displays. Atari designed into their machine as much software-controlled hardware flexibility as possible. In this way they hoped to achieve widely varied effects without changing the basic hardware.

We have covered a variety of playfield (i.e. display list generated) graphics and gotten familiar with player-missile graphics. Now we will cover another of the many playfield features, the ability to redefine a character set.

A character set is the table of shapes the Atari uses to define each character. This character set, or shape table is what makes an "A" character look different from a "B" on the screen. With the Atari, these shapes may be altered at will.

With most computers, you cannot change the characters the designers give you. The shapes are stored in ROM and cannot be modified except by creating a new ROM, a task beyond most of us. This places a limitation on those machines, for reprogramming character shapes is a powerful tool for certain applications.

If we are writing a program to teach the Russian language, we would naturally like to be able to write words in that language. But Russian has characters not found in English. With most machines, you are stuck at this point. Unless you use slow and clumsy high resolution graphics to draw characters, you cannot use the Russian characters.

On the Atari, it is easy to design your own characters. You can use new letters for the Russian lesson, and save yourself a lot of time and effort.

If you need some small figures on a character screen, but do not want to worry about mixing graphics modes, a character set might solve your problem. You can control dots the size of an individual graphics 8 pixel with custom characters, for that is the size dot characters are built from. You can even mix those special symbols in with your other text. For mathematicians needing special characters such as summation and integral characters, this could be a real help.

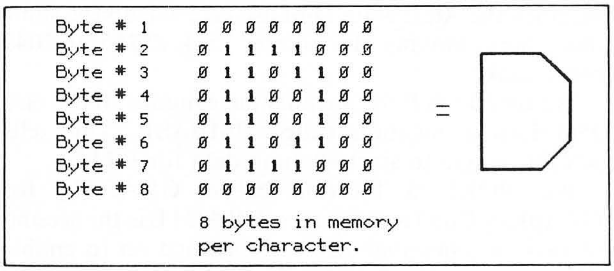
As soon as you begin to consider characters as graphics 8 figures drawn at high speed on the screen, more and more interesting possibilities will occur to you for the use of reprogrammed characters. We will review a bit about character shapes and generation, then learn how to modify them.

Character Shapes

The Atari plots letters and graphics on the screen using individual TV dots. It uses 320 horizontal dots and 192 scan lines for this purpose. Characters are 8 X 8 groups of dots, that is 320/8 or 40 characters across and 192/8 or 24 rows. There is no space on the screen between characters. Such space is provided for within the character shapes. This makes possible continuous script letters, which "flow" from one to the next with no interruption. It also enables screen graphics using characters that have no "breaks" in them.

Character shapes are stored as an 8 X 8 group of bits. A lit dot is represented by a "1" bit, an unlit dot by a "0" bit (Figure 25). Since each horizontal "slice" of the character is 8 bits, the Atari's designers put each slice into one byte, for a total of eight bytes per character. There are 128 different possible characters, and they are stored all grouped together, so the complete "character set" is 128 X 8 or 1024 bytes long. (Figures 26 and 27).

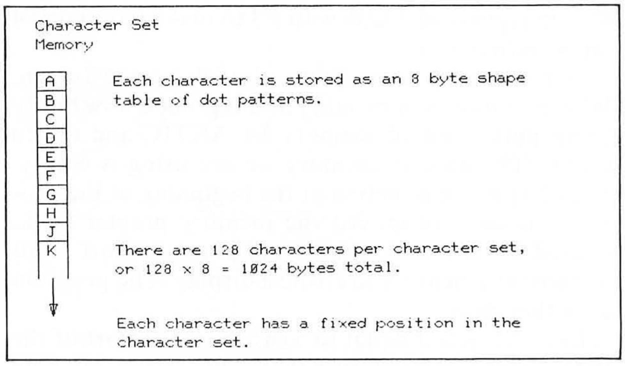
Every time a character is displayed, the Atari consults this table.



```
Byte * 1     Ø Ø Ø Ø Ø Ø Ø Ø
Byte * 2     Ø 1 1 1 1 Ø Ø Ø
Byte * 3     Ø 1 1 Ø 1 1 Ø Ø
Byte * 4     Ø 1 1 Ø Ø 1 Ø Ø
Byte * 5     Ø 1 1 Ø Ø 1 Ø Ø    =
Byte * 6     Ø 1 1 Ø 1 1 Ø Ø
Byte * 7     Ø 1 1 1 1 Ø Ø Ø
Byte * 8     Ø Ø Ø Ø Ø Ø Ø Ø

             8 bytes in memory
             per character.
```

Figure 25.



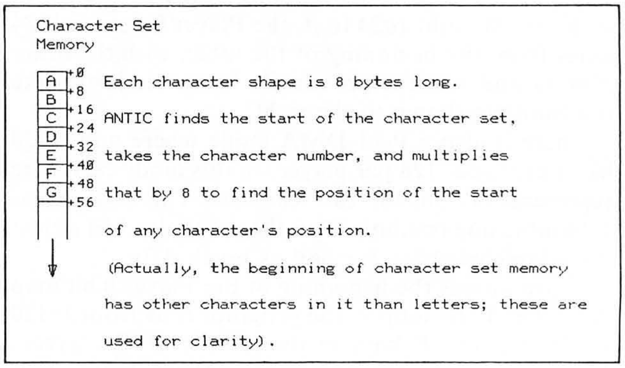
Character Set
Memory

Each character is stored as an 8 byte shape table of dot patterns.

There are 128 characters per character set, or 128 x 8 = 1Ø24 bytes total.

Each character has a fixed position in the character set.

Figure 26.



Character Set
Memory

Each character shape is 8 bytes long.

ANTIC finds the start of the character set, takes the character number, and multiplies that by 8 to find the position of the start of any character's position.

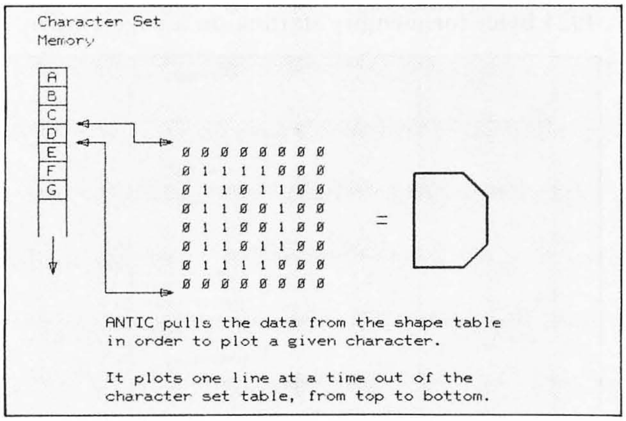
(Actually, the beginning of character set memory has other characters in it than letters; these are used for clarity).

Figure 27.

When ANTIC finds a display list entry to generate characters (modes 0, 1 and 2 to Basic users), it looks to the current location in display memory, kept in an internal ANTIC register. Let's assume graphics 0. One graphics 0 instruction means 40 characters are plotted in one row for one display block. In a character mode, one byte of display memory represents one character, so ANTIC fetches 40 bytes. Each character has a unique number, 0-127, and ANTIC uses that number to look up the character's shape in the character set.

First, ANTIC must find the character set. That is easy. The character set is sent to ANTIC every sixtieth of a second by the operating system as part of the screen refresh process. It is controlled by location 2F4 Hex or 756 decimal. This location we will call CHBAS, for "CHaracter set BASe". The number in this byte, when multiplied by 256, specifies the start of the character set in memory. In the Atari, like all 6502 processor machines, memory is divided up into "pages". Each page is 256 bytes long, exactly corresponding to 8 bits of address. In a 16 bit address, the upper eight bits specify which page number, and the lower 8 bits specify which byte within the page. Because the character set always starts on an even page mark, we only need to tell ANTIC where to find the character set's first page. Next, we must find the group of 8 bytes within the character set that represent the shape for an individual character.

The character number in display memory, known as the "internal character set number" (this is not ATASCII!) is multiplied by 8. This is then added to the CHBAS*256 number to give ANTIC the starting address in memory of the particular character's shape table. When displaying the character, ANTIC takes the first byte of the shape table, displays it as 8 on or off dots according to the bits in the shape table, then moves down one byte in the shape table for the next line. After eight passes, it has moved down 8 scan lines and read 8 bytes and is finished with the character (Figure 28).



```
Ø Ø Ø Ø Ø Ø Ø Ø
Ø 1 1 1 1 Ø Ø Ø
Ø 1 1 Ø 1 1 Ø Ø
Ø 1 1 Ø Ø 1 Ø Ø
Ø 1 1 Ø Ø 1 Ø Ø
Ø 1 1 Ø 1 1 Ø Ø
Ø 1 1 1 1 Ø Ø Ø
Ø Ø Ø Ø Ø Ø Ø Ø
```

ANTIC pulls the data from the shape table in order to plot a given character.

It plots one line at a time out of the character set table, from top to bottom.

Figure 28.

If we tell ANTIC the shape table began somewhere else in memory, it would look to the new location and start using whatever data was there to display characters. You will recall that earlier we told ANTIC that display memory was located in low memory, to watch it display pages 0 and 1 of memory, an area where there is all sorts of activity going on, as characters. This is the same idea. If the new area of memory happens to be a table of character shapes, redefined to what we want them to be, ANTIC will use them without complaint.

We cannot change the existing character set. It is stored in ROM (read-only-memory) and cannot be modified. So what we need to do is copy that ROM character set into RAM (Read-Write-Memory), where we can modify it, and then tell ANTIC to start looking to RAM for the character set. All we do (to change where ANTIC looks) is POKE a new page number in memory into location 756. A sixtieth of a second later, the operating system will give ANTIC that new value as part of the screen refresh, and it will start using it.

Our demonstration programs will demonstrate this process and show us how characters are stored.

Program 21 begins at the start of the unmodifiable character set the Atari normally uses, the ROM character set. It fetches 8 bytes per character, breaks each byte up into individual bits, and displays them as "0"'s and "1"'s. The program goes through the entire character set this way, displaying what the characters look like in binary patterns. See the listing for an example. You can see how ANTIC uses the "1" bits to plot lit dots and thus characters.

```
80 DIM BIN$(8)
90 REM O.S. SHADOW FOR CHBAS=2F4 HEX
100 CH=2*256+15*16+4
130 CHBAS=PEEK(CH)*256
200 REM
210 FOR CHNUM=0 TO`127
211 PRINT CHNUM,CHR$(CHNUM)
212 GOSUB`220
213 PRINT
214 NEXT CHNUM
215 REM FIDDLE CHR$ VALUE TO ROM VAL
220 IF CHNUM<32 THEN CH=CHNUM+64
230 IF CHNUM<96 THEN IF CHNUM>31 THEN CH=CHNUM-32
240 IF CHNUM>95 THEN CH=CHNUM
250 REM PULL 8 BYTES, TRANSLATE,PRINT
260 CLOC=CHBAS+(8*CH)
270 FOR B=0 TO 7
280 BYTE=PEEK(CLOC+B)
290 GOSUB 500
300 PRINT B+1;"* ";BIN$
310 NEXT B
320 RETURN
500 REM DECIMAL TO BINARY
505 BIN$="        "
510 DIV=128
515 BYTE1=BYTE
520 FOR T=1 TO 8
530 BIT=INT(BYTE1/DIV)
535 IF BIT=1 THEN BIN$(T,T)="1"
540 IF BIT=1 THEN BYTE1=BYTE1-DIV
550 DIV=INT(DIV/2)
560 NEXT T
610 RETURN
```

Program 21.

You are going to soon notice that characters are not stored in ATASCII order. They are in the order of the internal character set, which is a different thing. You can find a listing of the internal order on page 55 of your Basic manual.

Program 22 dumps the specified character to the printer; just type in the letter whose bit pattern you would like to be displayed. It is converted into an ATASCII number, then into the internal character set number, then displayed. This program is handy in showing how to convert from ATASCII to internal format. To find the right bytes in the character set, the internal number is just multiplied by 8 and added to the number that represents the start of the character set, which you will recall is just how ANTIC does it.

```
80 DIM BIN$(8)
90 REM O.S. SHADOW FOR CHBAS=2F4 HEX
100 CH=2*256+15*16+4
130 CHBAS=PEEK(CH)*256
200 PRINT "ENTER CHARACTER NUMBER"
210 INPUT CHNUM
211 PRINT CHNUM,CHR$(CHNUM)
212 GOSUB 220
213 PRINT
214 GOTO 200
215 REM FIDDLE CHR$ VALUE TO ROM VAL
220 IF CHNUM<32 THEN CH=CHNUM+64
230 IF CHNUM<96 THEN IF CHNUM>31 THEN CH=CHNUM-32
240 IF CHNUM>95 THEN CH=CHNUM
250 REM PULL 8 BYTES, TRANSLATE,PRINT
260 CLOC=CHBAS+(8*CH)
270 FOR B=0 TO 7
280 BYTE=PEEK(CLOC+B)
290 GOSUB 500
300 PRINT B+1;"* ";BIN$
310 NEXT B
320 RETURN
500 REM DECIMAL TO BINARY
505 BIN$="        "
510 DIV=128
515 BYTE1=BYTE
520 FOR T=1 TO 8
530 BIT=INT(BYTE1/DIV)
535 IF BIT=1 THEN BIN$(T,T)="1"
540 IF BIT=1 THEN BYTE1=BYTE1-DIV
550 DIV=INT(DIV/2)
560 NEXT T
610 RETURN
```

Program 22.

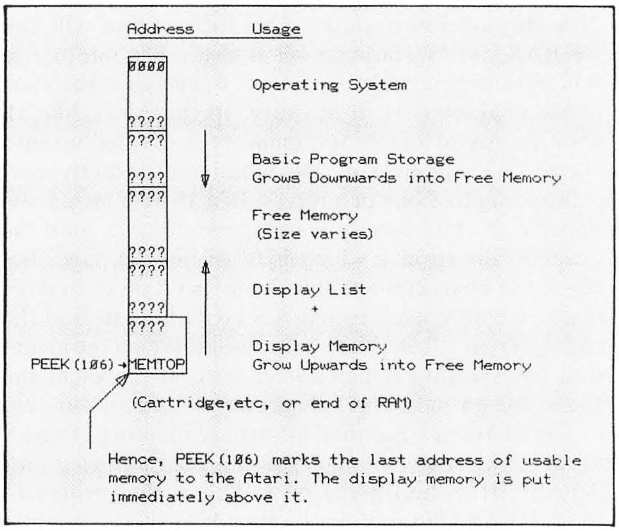
The character set we are currently looking at is in ROM, as previously mentioned. Let's learn how to move it to RAM to allow us to modify it. This will consist of three steps:

1. Finding a place to put it. We need 1024 free contiguous bytes of RAM.
2. Copying the ROM character set to RAM.
3. Changing the "pointer" ANTIC uses to find the character set from its old ROM location to the new RAM location.

Step 1 is tricky. To properly understand how to do this, we must delve into some Atari memory secrets.

When the Atari is first turned on, a check is made to determine where RAM ends. This can be anywhere from 8K to 48K from the beginning of memory; it depends on how many memory boards you have installed. In location 106 decimal (6A hex) is stored the page number of the first byte of non-existent memory. In other words, 256*PEEK(106) is the address of the first byte of nonexistent memory.

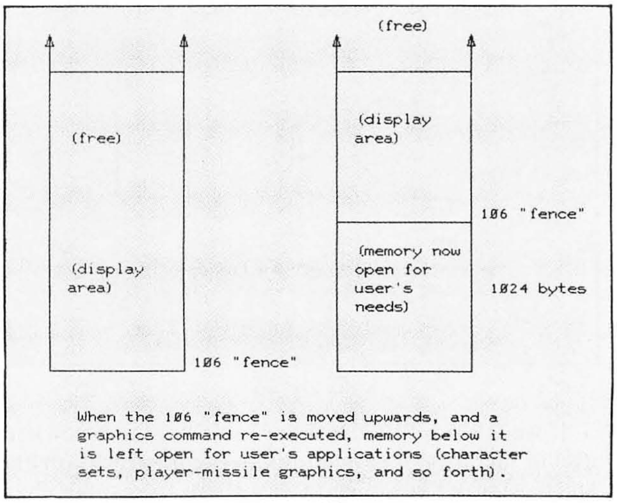
Now the Atari uses the very top of RAM memory, wherever that might be, for the display memory and display list storage. Right below that is free RAM, and below that is Basic storage. (Basic and the graphics modes "grow" towards each other into free RAM when they use more memory). So whenever a graphics command is executed, and the Atari needs to set up a new display memory-display list, it checks location 106 to see where RAM ends. It then backs up the required number of locations and puts the display memory in (Figure 29). Think of memory location 106 as the Atari's "fence", used to find the end of memory.



Hence, PEEK(106) marks the last address of usable memory to the Atari. The display memory is put immediately above it.

Figure 29.

Now let's assume we POKE 106,PEEK(106)—4). This will move back the end of memory fence by 4 pages. Each page, you will recall, is 256 bytes, so that is 4 times 256 or 1024 bytes moved back. We then execute a graphics command, so the Atari will move the display memory list out of that 1024 byte area, behind our fence (Figure 30). In this way we reserve 1024 bytes for memory starting on a page border.



When the 106 "fence" is moved upwards, and a graphics command re-executed, memory below it is left open for user's applications (character sets, player-missile graphics, and so forth).

Figure 30.

There are several advantages to getting 1024 bytes this way. It does not matter what size memory machine you have, as long as the minimum 1024 bytes are available. It does not matter how long your Basic program is or what graphics mode you are in. You can see it is quite a handy general purpose thing to have.

This is also the preferred technique to use when reserving memory for the Player-Missile bitmap area. 8 pages are required for a 2048 byte bitmap (single line resolution) or 4 for 1024 bytes (double line resolution). You will see this byte 106 modification in most articles on Player-Missile graphics.

We now know the beginning of the RAM area, and where the ROM character set starts (E000 Hex or 57344 Decimal). Let's copy the ROM character set to RAM (Program 23). This program moves the 106 pointer back 4 pages and copies the character set over. It takes a while; around ten seconds is needed to copy 1024 bytes. Basic just is not very fast at copying data.

```
60 REM COPIES CHARSET TO RAM
100 MEMTOP=PEEK(106)
110 GRTOP=MEMTOP-4
120 POKE 106,GRTOP
130 REM RESET
140 GRAPHICS 0
141 LIST
160 CHROM=PEEK(756)*256
170 CHRAM=GRTOP*256
180 PRINT "COPYING."
500 FOR N=0 TO 1023
510 POKE CHRAM+N,PEEK(CHROM+N)
520 NEXT N
530 PRINT "COPIED."
535 REM NOW MODIFY POINTER
540 POKE 756,GRTOP
```

Program 23.

Finally, the CHBAS pointer is changed to reflect the page of the beginning of our new RAM area. ANTIC is now using the RAM character set (Figure 31).

Program 23 is not going to show you much, for ANTIC will still be displaying characters as usual. So let's watch the copy process in action. This time we will move the character set pointer first, then do the copy. Your screen will suddenly start displaying whatever junk is in memory at the start of the copy as the pointer is changed, then more and more letters will appear as Basic gets more and more character shapes copied into the RAM table. At the end of the copy, the screen will once again appear normal (Program 24).

Program 25 presents an interesting variation. It copies characters from ROM and RAM upside down. It does this by copying the eighth byte of every character into the first byte of that character's new bitmap, the seventh to the second, and so forth. The result is that the new RAM bitmap is an inverted image of the ROM bitmap. This is a lot of fun. The characters will still be on the screen, and you can even edit them. They will just be upside down.

Program 26 shows another useful variation. It makes every character's last byte be a 255, or solid 1's. This puts a solid line at the base of the characters, and there is thus a line at the bottom of each of the 24 character rows. If you have been wondering how to underline a particularly important concept on the Atari screen, you have just found out how.

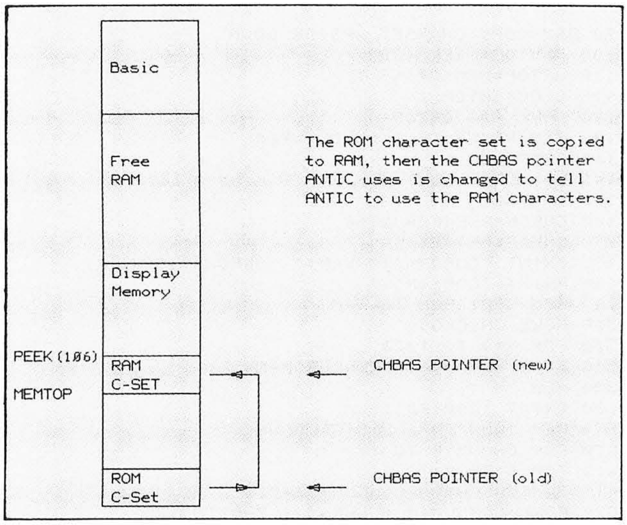
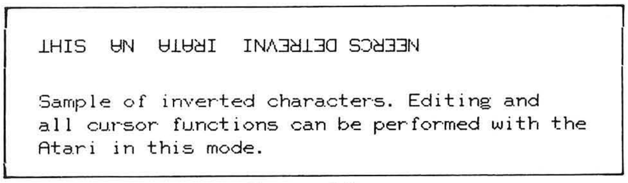

Basic
Free RAM
Display Memory
PEEK(106)
MEMTOP
RAM C-SET
ROM C-Set
The ROM character set is copied to RAM, then the CHBAS pointer ANTIC uses is changed to tell ANTIC to use the RAM characters.
CHBAS POINTER (new)
CHBAS POINTER (old)

Figure 31.



THIS AN ATARI INVERTED SCREEN

Sample of inverted characters. Editing and all cursor functions can be performed with the Atari in this mode.

Figure 32.

Program 27 illustrates another handy character set feature. We can POKE different values into the CHBAS pointer and thus switch between multiple character sets immediately. In program 27 we have two character sets, one normal, one flipped upside down. The program switches between them rapidly for an effect that is hard on the eyes. Assembly language programmers take note: with a display list interrupt, you can change character sets midway down the screen. The possibilities with that are amazing. Just POKE a new value into the ANTIC hardware address for CHBAS.

Now let's assume we have decided to modify a ROM character set to accustom one of our needs. Let's work it out by hand the first time. Incidentally, an editor based on this hand working out is not too difficult to write, and there are many out on the market. None however, have the storage scheme that we will be discussing shortly.

```
60 REM COPIES CHARSET TO RAM
100 MEMTOP=PEEK(106)
110 GRTOP=MEMTOP-4
120 POKE 106,GRTOP
130 REM'RESET
140 GRAPHICS 0
141 LIST
160 CHROM=PEEK(756)*256
170 CHRAM=GRTOP*256
172 REM NOW MODIFY POINTER
173 POKE 756,GRTOP
180 PRINT "COPYING."
500 FOR N=0 TO 1023
510 POKE CHRAM+N,PEEK(CHROM+N)
520 NEXT N
530 PRINT "COPIED."
```

Program 24.

```
50 REM COPY CHARSET UPSIDE DOWN
100 MEMTOP=PEEK(106)
110 GRTOP=MEMTOP-4
115 CLOC=GRTOP
120 POKE 106,GRTOP
130 REM RESET GR.0 DM/DL AREA
140 GRAPHICS 0
141 LIST
150 CH=756
160 CHROM=PEEK(CH)*256
170 CHRAM=GRTOP*256
175 PRINT "CHRAM=";CHRAM;"  CHROM=";CHROM
180 PRINT "COPYING."
190 REM COPY ROM TO RAM
300 POKE CH,CLOC
500 FOR N=0 TO 1023
510 POKE CHRAM+N,PEEK(CHROM+N)
520 NEXT N
530 PRINT "COPIED."
550 REM  NOW COPY UPSIDE DOWN
600 FOR CHNUM=0 TO 127
610 FOR BYTE=0 TO 7
615 Z=PEEK(CHROM+(CHNUM*8)+BYTE)
620 POKE (CHNUM*8)+(CHRAM)+(7-BYTE),Z
630 NEXT BYTE
635 NEXT CHNUM
640 PRINT "RECOPIED."
```

Program 25.

```
100 MEMTOP=PEEK(106)
110 GRTOP=MEMTOP-4
115 CLOC=GRTOP
120 POKE 106,GRTOP
130 REM RESET GR.0 DM/DL AREA
140 GRAPHICS 0
141 LIST
150 CH=756
160 CHROM=PEEK(CH)*256
170 CHRAM=GRTOP*256
175 PRINT "CHRAM=";CHRAM;"  CHROM=";CHROM
180 POKE CH,GRTOP
600 FOR CHNUM=0 TO 127
610 FOR BYTE=0 TO 7
615 Z=PEEK(CHROM+(CHNUM*8)+BYTE)
616 IF BYTE=7 THEN LET Z=255
620 POKE (CHNUM*8)+(CHRAM)+(BYTE),Z
630 NEXT BYTE
635 NEXT CHNUM
640 PRINT "RECOPIED."
```

Program 26.

First, let's design the character we want as an 8 X 8 dot matrix

```
0 0 1 1 1 1 0 0
0 1 0 0 0 0 1 0
1 0 1 0 0 1 0 1
1 0 0 0 0 0 0 1
1 0 1 0 0 1 0 1
1 0 0 1 1 0 0 1
0 1 0 0 0 0 1 0
0 0 1 1 1 1 0 0
```

This is, of course, the character from the "Have a Nice Day!" button.

Let's determine the bit patterns. You can do this by either converting each nibble (4 bits) to hex and then going to decimal, or for those of you without binary experience, just add the number shown on the top of the column to the total for that line whenever the dot it represents is on. For example, in the diagram, 16 and 8 are "on", so add 16 + 8 = 24.

```
45 REM THEN FLIPS BACK AND FORTH
100 MEMTOP=PEEK(106)
110 GRTOP=MEMTOP-4
115 CLOC=GRTOP
120 POKE 106,GRTOP
130 REM RESET GR.0 DM/DL AREA
140 GRAPHICS 0
141 LIST
150 CH=756
160 CHROM=PEEK(CH)*256
170 CHRAM=GRTOP*256
175 PRINT "CHRAM=";CHRAM;"  CHROM=";CHROM
180 PRINT "COPYING."
190 REM COPY ROM TO RAM
300 POKE CH,CLOC
500 FOR N=0 TO 1023
510 POKE CHRAM+N,PEEK(CHROM+N)
520 NEXT N
530 PRINT "COPIED."
550 REM  NOW COPY UPSIDE DOWN
600 FOR CHNUM=0 TO 127
610 FOR BYTE=0 TO 7
615 Z=PEEK(CHROM+(CHNUM*8)+BYTE)
620 POKE (CHNUM*8)+(CHRAM)+(7-BYTE),Z
630 NEXT BYTE
635 NEXT CHNUM
640 PRINT "RECOPIED."
700 REM FLIP
710 POKE CH,224:REM NORMAL ROM
720 POKE CH,CLOC
730 GOTO 710
```

Program 27.

At the end of this process, you will have 8 bytes of data which represent the bitmap for that character. Next, let's figure out which character we are going to replace with our SMILE character. How about the space character? There are plenty of those on the screen. The space character is the first one in the ROM-RAM character set, character number 0, in internal code. So what we do is POKE these 8 bytes into the location where the space character's bitmap is located, replacing them with the SMILE character. See program 28, which is just our routine to copy the character set from ROM to RAM with the added POKEs (the numbers are in the DATA statement).

If we wanted to replace another character, we would multiply its character number by 8, add that number to the address of the start of the character set, and start POKEing there. That is why "LOC=(CHBAS + (8+0))" was used. Replace the 0 with whatever number you wish.

At this point your Atari will be smiling proudly at you from everyplace a space used to be. Take a minute to enjoy the happiness of your success.

Storing & Retrieving Your Character Set

You do not have to re-POKE your character set each time you want to use it. After all, the POKE method of copying the 1024 bytes from ROM to RAM is one of the greatest sleep inducers known. Let's solve all these problems with some custom routines for character set work. They all work off of string manipulations, which are among the most powerful and usable on the Atari. The reason for their power is their speed in an otherwise slow Basic; the string manipulation routines are just high speed assembly language copy routines. Let's subvert them to our purposes, and have assembly speed without all the hassles.

Each string is stored in memory as a continuous group of bytes. A string has a DIMensioned length, a "currently in use" length, and a location in memory. Let's assume they both have length 1024. And let's assume that the storage location where the Atari thinks RAM$ is in memory just happens to be our RAM character set area. Let's further assume that ROM$ is in the ROM character set area (or so the Atari thinks). What will happen when we then execute RAM$=ROM$?

The Basic string manipulation routines will copy 1024 bytes (dimensioned length) from ROM$ to RAM$, and thus copy the ROM charset to the RAM charset at extremely high speed!

You can modify the RAM character set any way you wish. Bear in mind you can do this with either a POKE or a string operator; when you modify the string, you are modifying the RAM character set. (You cannot modify ROM$.) Let's write RAM$ out to disk. The Atari will store your character set out on disk as a string. Let's read it back in at some later date, still using all string manipulation operators, and store it back into the character set area. You will have just stored and recovered your character set. No hassles with bits and bytes, just a PRINT to disk and an INPUT later on.

The power of the copy capability is also usable in player-missile graphics. You can assign a string to the player bitmap area, and then move the player up and down at high speed using a $=$ operation. This is a nice fast way to move a player vertically, which before required either assembly language or slow POKE copies. And strings may be used for data storage. The display list interrupt routine listed earlier used a string to store data bytes for color registers, and another string to hold the assembly program used for the interrupt handling.

Let's learn how to change where the Atari thinks a string is located in memory. Then we will get to the actual subroutines you can use.

The Atari keeps two tables in memory for Basic (among others) that deal with string variables. One is called the variable table, the other the array table. There are 128 possible variable names on the Atari, numbered 0-127, and the variable table has an 8 byte entry for each name in use. All the entries are packed together. For strings this entry has dimensioned and in-use length, and where in the array table the string is stored. The array table is the other table. In it the string's actual data is kept. So, what we have to do is alter the dimensioned and in-use length as shown in the variable table, both to 1024, then modify where the Atari thinks the variable is stored in the array table. The only tricky part to this is that the address of where the string is actually stored is relative to the array table; in other words, a "0" for this value does not mean the string starts at location 0, it starts at the beginning of the array table.

You can find the beginning of the variable table by:
VT=PEEK(134)+256*PEEK(135)

The beginning of the array table is found by:
AT=PEEK(140)+256*peek(141)

We will examine the actual layout of the variable table entries assuming that RAM$ and ROM$ are the first two variables in the variable table. In reality to do this they must be the first variables types in a NEW program or ENTERed from a program LISTed to disk. (A SAVE-LOAD will not work, it stores the variable table along with the program.) So if you're starting out with a new program, just have the DIM line (10 DIM RAM$(1),ROM$(1) as the first line of your program after typing NEW; if you are adding these to an existing program, make sure that the first line and LIST it to disk and ENTER back to rewrite the tables.

The variable table entry is created for any variable referenced by your program. This includes variables you used once and then deleted; they are still there taking up space. You can run out of space in the variable table when it gets too full of these nonexistent variables. LIST, then ENTER from disk forces a new variable table to be built.

Here's the variable table with explanations.

Location	Value	Meaning
VT+0	129	"This is a string"
VT+1	0	"This is variable #0"
VT+2, VT+3	??	16 bits. Location from the start of AT.
VT+4, VT+5	??	DIMensioned length.
VT+6, VT+7	??	In-use length

This is the entry for RAM$, the first string in the table. The entry for ROM$ immediately follows.

This subroutine should now become clear. It modifies the address and length of RAM$ to that of the character set. It not only copies ROM$ to RAM$, it also modifies the variable table data for ROM$. (All the modifying, by the way, is quite speedy, so the RAM$=ROM$ still executes much faster than the previous POKE copy). (See Program 29).

```
60 REM COPIES CHARSET TO RAM
70 REM POKES POINTER B/4 COPY
80 REM ADDS SMILE
100 MEMTOP=PEEK(106)
110 GRTOP=MEMTOP-4
120 POKE 106,GRTOP
130 REM RESET
140 GRAPHICS 0
141 LIST
145 CHROM=PEEK(756)*256
150 REM NOW MODIFY POINTER
160 POKE 756,GRTOP
170 CHRAM=GRTOP*256
180 PRINT "COPYING."
500 FOR N=0 TO 1023
510 POKE CHRAM+N,PEEK(CHROM+N)
520 NEXT N
530 PRINT "COPIED."
540 REM ABCDEFGHIJKLMNOPQRSTUVWXYZ
550 REM 123567890!"#$%&'@()<>-=+*
1000 REM SMILE BUTTON LAYOUT:
1010 REM 00000000 00 00
1020 REM 01100110 66 102
1030 REM 01100110 66 102
1040 REM 00000000 00 000
1050 REM 01000010 42 66
1060 REM 00111100 3C 60
1070 REM 00011000 18 24
1080 REM 00000000 00 00
1089 REM
1090 DATA 00,102,102,000,66,60,24,00
1100 FOR ADDR=CHRAM TO CHRAM+7
1110 READ DAT:POKE ADDR,DAT
1120 NEXT ADDR
```

Program 28.

```
5 REM PROGRAM TO COPY ROM TO RAM
6 REM USING STRING MANIPULATORS
7 REM
8 REM NOTE MOST CALCULATIONS ARE NOT
9 REM HARDCODED TO ALLOW OTHER USE
10 DIM RAM$(1),ROM$(1):REM VT ENTRY 1
90 REM GET ARRAY,VARIABLE,DL,DM LOC
105 AT=PEEK(140)+256*PEEK(141)
110 VT=PEEK(134)+256*PEEK(135)
120 POKE 106,PEEK(106)-16:REM 4K MOVE
125 GRAPHICS 0:REM RESET OUT OF TOP AREA
130 RAMLOC=PEEK(106)*256
150 REM CALCULATE OFFSET FROM AT
160 OFFRAM=RAMLOC-AT
170 OFFROM=(14*4096)-AT
220 REM CALCULATE LO,HI BYTES
225 LENS=1025:REM C-SET LENGTH
230 LENHI=INT(LENS/256)
240 LENLO=INT(LENS-(LENHI*256))
245 REM
250 OFFRAMH=INT(OFFRAM/256)
260 OFFRAML=INT(OFFRAM-(256*OFFRAMH))
270 OFFROMH=INT(OFFROM/256)
280 OFFROML=INT(OFFROM-(256*OFFROMH))
300 REM REWRITE RAM$ DATA'IN VT
310 REM VT+0 = 129
320 REM VT+1 = 0   (VAR #0)
330 POKE VT+2,OFFRAML:REM OFFSET
340 POKE VT+3,OFFRAMH:REM OFFSET
350 POKE VT+4,LENLO:REM DIM LENGTH
360 POKE VT+5,LENHI:REM DIM LENGTH
370 POKE VT+6,LENLO:REM USED LENGTH
380 POKE VT+7,LENHI:REM USED LENGTH
400 REM REWRITE ROM$ DATA IN VT
410 REM VT+8 = 129
420 REM VT+9 = 1   (VAR #0)
430 POKE VT+10,OFFROML:REM OFFSET
440 POKE VT+11,OFFROMH:REM OFFSET
450 POKE VT+12,LENLO:REM DIM LENGTH
460 POKE VT+13,LENHI:REM DIM LENGTH
470 POKE VT+14,LENLO:REM USED LENGTH
480 POKE VT+15,LENHI:REM USED LENGTH
500 REM RESTORE CHBAS POINTER
510 POKE 756,PEEK(106)
515 REM NOW DO COPY.
520 RAM$=ROM$
```

Program 29.

Part II
Graphics Tips

Part II
Graphics Tips

Design Philosophy and GTIA Demos

This last year marks a period of incredible growth for the Atari computer. The Atari Program Exchange got going and is now shipping a large volume of Atari software. This exchange provides low-cost but relatively high quality software written by outside users.

Last year at this time there was no *Jawbreakers, Asteroids,* or *Missile Command*; now these programs are being surpassed (Have you seen *Mouskattack* yet? The cover art alone is worth the purchase price.)

Last year we had the Basic and Assembler cartridges, neither of which was designed for speed or for large programs. Now we have Microsoft Basic, the awesome new Editor/Macro Assembler, Pilot, Forth, Lisp, and according to a letter I just received, Algol.

The tools to develop high quality software are now available; and it is a safe bet that more good software will be appearing on the market.

Outside manufacturers are also producing a wide variety of hardware for the Atari. There are several modems available, one of which I will review shortly (the direct-connect Microconnection).

There is also a good deal of hobbyist-oriented equipment, such as EPROM burners, I/O port connectors, and whatnot, and a light pen is now available from non-Atari sources.

Percom Corp. in Texas is now marketing an entire line of Atari-compatible single, double and quad density disk drives, and other companies have introduced Winchester drives. Atari has even published a list of outside vendors of software and hardware for their machine.

And did I mention the news that Atari has surpassed Apple in sales? And rumors of new Atari machines are currently flying (something may be announced in June, I am told). This machine is here to stay, and will continue to grow for quite some time.

What Is an Atari?

If you own an Atari, you may be curious to learn what you have bought. I searched long and hard for a definition of what Atari has produced; "home computer" is too vague. After all, it does have a specific and limited place in the market. For example Atari dropped the 815 double density dual disk drive when it did not conform to their definition of a "home computer."

The definition I finally found, is a "home computing appliance." Chew on those words for a while; they contain the essence of the design and marketing strategy of the Atari. This is a second-generation consumer-oriented machine, carefully designed and oriented towards the home market from the ground up.

Atari is not in the business market. They have no intention of competing with Tandy, Apple, and IBM, who are currently beating their respective heads together trying to capture that field. Atari stands with Texas Instruments in the home computer marketplace with a strategy of aiming at the home user.

Products designed for home use—principally games, home finance, and education/development—are released and pushed hard. Pilot offers a good example of the educational potential of this machine, and the games available for the Atari are becoming the standard for the home computer industry.

Bear in mind also that this is a consumer machine. It is not designed for a hardware or computer professional. It is designed for an average person who wants a "home computing appliance."

It is not as hardware oriented as the Apple. It doesn't have a collection of open "slots" on the various busses available. What housewife seeking help with her checkbook honestly cares whether or not she can access the interrupt request line with a plug-in card?

The layout is for a consumer. The machine is attractively styled, and goof-proof. Memory and the Operating System are packaged in cartridges located under the front cover, and joysticks plug in easily under the front keyboard.

Many of the programs available can be used by a consumer with little knowledge of computers; they plug in on ROM cartridges (most four-year old kids I know can master this trick), and the system Reset key is protected against accidental press, and so on.

I remember being told that "The Atari isn't a serious machine because it doesn't offer PASCAL." I have heard this sort of complaint many times, always directed at something the Atari lacked, be it Cobol, Fortran-1933, or whatever. The people who voice these objections (in most cases, computing professionals) don't understand that the Atari home computer is Atari's very serious attempt to make a computer that a home user can get along with—not necessarily a programmer, just a home user. It may not have the current languages that are in vogue today (and possibly gone tomorrow). But this is by design and not by default.

So, while the Atari may not at first glance *look* like a powerful computing machine, with lots of lights, integrated circuits, and cables all over the place, it is. It has just been designed for a home computer user according to Atari's idea of who the home computer user is.

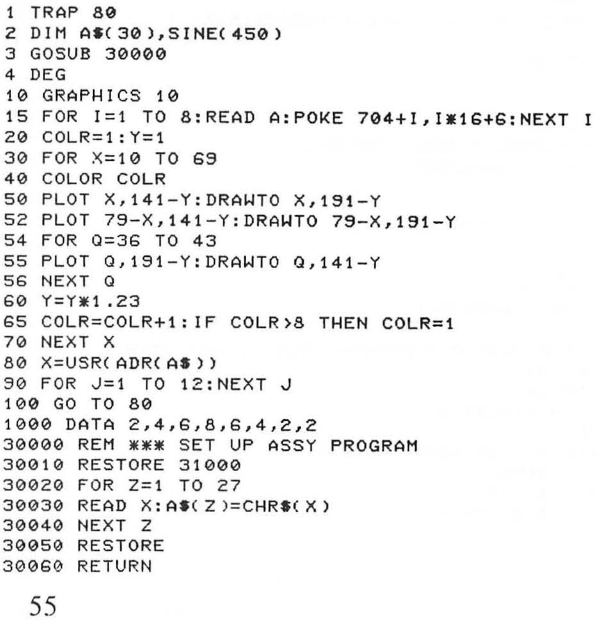
Oh, and yes, Pascal is now available for the Atari.

GTIA Demonstration Programs

As of January 1982, Atari began shipping all Atari 800 units with GTIA graphics chips. The GTIA chip replaces the CTIA graphics chip and allows three more graphics modes. Don't worry, the operating system ROM and the Basic cartridge were written with GTIA in mind. GTIA is a superset of the CTIA functions.

Several short demonstration programs for the GTIA exist. I don't know where

LOGO



```
1 TRAP 80
2 DIM A$( 30 ),SINE( 450 )
3 GOSUB 30000
4 DEG
10 GRAPHICS 10
15 FOR I=1 TO 8:READ A:POKE 704+I,I*16+6:NEXT I
20 COLR=1:Y=1
30 FOR X=10 TO 69
40 COLOR COLR
50 PLOT X,141-Y:DRAWTO X,191-Y
52 PLOT 79-X,141-Y:DRAWTO 79-X,191-Y
54 FOR Q=36 TO 43
55 PLOT Q,191-Y:DRAWTO Q,141-Y
56 NEXT Q
60 Y=Y*1.23
65 COLR=COLR+1:IF COLR>8 THEN COLR=1
70 NEXT X
80 X=USR( ADR( A$ ) )
90 FOR J=1 TO 12:NEXT J
100 GO TO 80
1000 DATA 2,4,6,8,6,4,2,2
30000 REM *** SET UP ASSY PROGRAM
30010 RESTORE 31000
30020 FOR Z=1 TO 27
30030 READ X:A$( Z )=CHR$( X )
30040 NEXT Z
30050 RESTORE
30060 RETURN
```

they were written, but I would assume somewhere at Atari by someone with a preliminary GTIA chip.

Below are several listings of sample GTIA programs. Users with CTIA graphics chips can try these programs, but don't expect spectacular results. Users with GTIA chips will be in for a pleasant surprise, indeed.

Feel free, as always, to modify them, and if you come up with a really neat effect, I'd appreciate a listing or a disk/tape (I'll return your disk/tape, of course.) The address appears at the beginning of the column.

By the way, if the author of these programs would care to step forward, I will certainly give him credit. I would like to know who wrote them.

Be sure to include the following two lines at the end of LOGO, HYPNO, ESCAPE, MELONS, SAS, and WHIRL.

31000 DATA 104, 162, 0, 172, 193, 2, 189, 194, 2, 157, 193, 2, 232, 224, 8, 144, 245, 140, 200

31010 DATA 2, 96, 65, 65, 65, 65, 65, 65 □

MELONS

```
2 DIM A$(30),SINE(450)
3 GOSUB 30000
4 DEG
9 FOR I=0 TO 90:? I:A=SIN(I):SINE(I)=A:SINE(180-I)=A:SINE(180+I)=-A:
  SINE(360-I)=-A:SINE(360+I)=A:NEXT I
10 GRAPHICS 10
15 RESTORE :FOR I=1 TO 8:READ A:POKE 704+I,A:NEXT I
18 FOR P=1 TO 2
20 Q=1
25 A=30
30 FOR ANG=180 TO 270 STEP 8
40 X=20*SINE(ANG+90)+25
50 Y=A*SINE(ANG)
55 Z=X:IF P=2 THEN Z=79-X
60 COLOR Q
65 IF ANG=180 THEN OLDX=Z:OLDY=Y
70 PLOT OLDX,96+OLDY
75 DRAWTO Z,96+Y
77 OLDX=Z:OLDY=Y
80 REM
90 NEXT ANG
95 Q=Q+1:IF Q>7 THEN Q=1
100 A=A-1
110 IF A>-30 THEN 30
200 FOR ANG=0 TO 180 STEP 10
210 COLOR 8
220 X=4*SINE(ANG+90)+25
225 Z=X:IF P=2 THEN Z=79-X
230 Y=30*SINE(ANG)
240 PLOT Z,96+Y
250 DRAWTO Z,96-Y
260 NEXT ANG
300 FOR I=1 TO 25
310 X=RND(0)*6+23
315 Z=X:IF P=2 THEN Z=79-X
320 Y=RND(0)*50+71
330 COLOR 0
340 PLOT Z,Y
350 NEXT I
400 NEXT P
900 REM X=USR(ADR(A$))
910 FOR J=1 TO 10:NEXT J
920 GO TO 900
1000 DATA 226,228,230,232,230,228,226,70
30000 REM *** SET UP ASSY PROGRAM
30010 RESTORE 31000
30020 FOR Z=1 TO 27
30030 READ X:A$(Z)=CHR$(X)
30040 NEXT Z
30050 RESTORE
30060 RETURN
```

HYPNO

```
2 DIM A$(30)
3 GOSUB 30000
10 GRAPHICS 10
15 FOR I=1 TO 8:POKE 704+I,(I-1)*32+22:NEXT I
20 Q=1
30 FOR Y=0 TO 191
40 COLOR Q
50 PLOT 0,Y
60 DRAWTO 79,191-Y
70 Q=Q+0.416666666:IF Q>8 THEN Q=1
75 REM FOR T=1 TO 100:NEXT T
80 NEXT Y
120 Q=1
130 FOR X=79 TO 0 STEP -1
140 COLOR Q
150 PLOT X,0
160 DRAWTO 79-X,191
170 Q=Q+1:IF Q>8 THEN Q=1
180 NEXT X
190 REM COLOR 0:PLOT 0,0:DRAWTO 79,191:PLOT 79,0:
    DRAWTO 0,191
200 X=USR(ADR(A$))
210 FOR J=1 TO 4:NEXT J
220 GO TO 200
30000 REM *** SET UP ASSY PROGRAM
30010 RESTORE 31000
30020 FOR Z=1 TO 27
30030 READ X:A$(Z)=CHR$(X)
30040 NEXT Z
30050 RESTORE
30060 RETURN
```

ESCAPE

```
2 DIM A$(30)
3 GOSUB 30000
10 GRAPHICS 10
15 FOR I=1 TO 8:READ A:POKE 704+I,A+224:NEXT I
17 Q=1
20 FOR I=0 TO 38
40 COLOR Q
42 X=I
45 Y=I*2
50 PLOT X,Y
60 DRAWTO 79-X,Y:PLOT X,Y+1:DRAWTO 79-X,Y+1
62 DRAWTO 79-X,190-Y
64 DRAWTO'X,190-Y:PLOT 79-X,190-Y+1:DRAWTO X,190-Y+1
66 DRAWTO X,Y
70 Q=Q+1:IF Q>8 THEN Q=1
80 NEXT I
100 X=USR(ADR(A$))
110 FOR J=1 TO 24:NEXT J
120 GO TO 100
1000 DATA 2,4,6,8,6,4,2,2
30000 REM *** SET UP ASSY PROGRAM
30010 RESTORE 31000
30020 FOR Z=1 TO 27
30030 READ X:A$(Z)=CHR$(X)
30040 NEXT Z
30050 RESTORE
30060 RETURN
```

SAS

```
2 DIM A$(30),SINE(450)
3 GOSUB 30000
4 DEG
8 ? "STAND BY"
9 FOR I=0 TO 90:? I:A=SIN(I):SINE(I)=A:SINE(180-I)=A:
  SINE(I+180)=-A:SINE(360-I)=-A:SINE(I+360)=A:NEXT I
10 GRAPHICS 10
15 RESTORE :FOR I=1 TO 8:READ A:POKE 704+I,A+224:NEXT I
19 C=1
20 FOR ANG=0 TO 359
30 X=40+30*SINE(ANG+90)
40 Y=96+80*SINE(ANG)
50 COLOR INT(C)
60 PLOT 65,96
70 DRAWTO X,Y
75 PLOT 65,95
77 DRAWTO X,Y-1
78 PLOT 65,94
79 DRAWTO X,Y-2
80 Q=USR(ADR(A$))
85 C=C+0.5:IF C>=9 THEN C=1
90 NEXT ANG
900 Q=USR(ADR(A$))
910 FOR I=1 TO 14:NEXT I
920 GO TO 900
1000 DATA 2,4,6,8,6,4,2,2
30000 REM *** SET UP ASSY PROGRAM
30010 RESTORE 31000
30020 FOR Z=1 TO 27
30030 READ X:A$(Z)=CHR$(X)
30040 NEXT Z
30050 RESTORE
30060 RETURN
```

BRASS

```
10 GRAPHICS 9
15 SETCOLOR 4,15,0
20 FOR Y=55 TO 0 STEP`-10
30 FOR X=0 TO 24
40 C=X:IF X>11 THEN C=24-X
45 C=C+3
50 Z=Y+(X)
55 D=INT(SQR(144-(X-12)*(X-12)))/2
57 COLOR 15-C
58 PLOT Z,Y+7-D
60 DRAWTO Z,Y+7+D
70 COLOR C
80 DRAWTO Z,180-Y+D
180 NEXT X
190 NEXT Y
200 GO TO 200
```

WHIRL

```
2 DIM A$(30),SINE(450)
3 GOSUB 30000
4 DEG
8 ? "STAND BY"
9 FOR I=0 TO 90:? I:A=SIN(I):SINE(I)=A:SINE(180-I)=A:
  SINE(I+180)=-A:SINE(360-I)=-A:SINE(I+360)=A:NEXT I
10 GRAPHICS 10
15 RESTORE :FOR I=1 TO 8:READ A:POKE 704+I,A+224:NEXT I
17 GOSUB 2000
20 FOR ANG=0 TO 359 STEP 2
25 Q=8
30 X=20*SINE(ANG+90)+40
40 Y=20*SINE(ANG)+96
42 COLOR 0:PLOT X,Y:IF ANG<180 THEN 90
43 AS=0
45 FOR W=1 TO 45
50 LOCATE X,Y+W,QW:IF QW=0 THEN AS=1
55 IF AS=1 THEN 80
57 COLOR Q
60 PLOT X,Y+W
70 Q=Q-1:IF Q<1 THEN Q=8
80 NEXT W
90 NEXT ANG
95 Z=20:U=20
175 Q=Q+1:IF Q>8 THEN Q=1
900 X=USR(ADR(A$))
910 FOR I=1 TO 8:NEXT I
920 GO TO 900
1000 DATA 2,4,6,8,6,4,2,2
2000 REM THIS IS THE HEART OF DIZZY
2020 Q=1
2030 FOR Y=0 TO 191
2040 COLOR Q
2050 PLOT 0,Y
2060 DRAWTO 79,191-Y
2070 Q=Q+0.416666666:IF Q>8 THEN Q=1
2080 NEXT Y
2130 FOR X=79 TO 0 STEP -1
2140 COLOR Q
2150 PLOT X,0
2160 DRAWTO 79-X,191
2170 Q=Q+1:IF Q>8 THEN Q=1
2180 NEXT X
2230 RETURN
30000 REM *** SET UP ASSY PROGRAM
30010 RESTORE 31000
30020 FOR Z=1 TO 27
30030 READ X:A$(Z)=CHR$(X)
30040 NEXT Z
30050 RESTORE
30060 RETURN
```

RAINBOW

```
100 REM GTIA TEST
115 GRAPHICS 10:FOR Z=704 TO 712:READ R:POKE Z,R:NEXT Z
116 DATA 0,26,42,58,74,90,106,122,138,154
130 FOR X=1 TO 8:COLOR X:POKE 765,X
140 PLOT X*4+5,0:DRAWTO X*4+5,159:PLOT X*4+1,159:POSITION X*4+1,0:XIO 18,#6,0,0,
"S:"
150 NEXT X
230 FOR X=8 TO 15:COLOR 16-X:POKE 765,16-X
240 PLOT X*4+5,0:DRAWTO X*4+5,159:PLOT X*4+1,159:POSITION X*4+1,0:XIO 18,#6,0,0,
"S:"
250 NEXT X
300 COLOR 0:PLOT 65,159:DRAWTO 0,159
400 FOR X=1 TO 8:Z=PEEK(704+X):Z=Z+16:IF Z>255 THEN Z=26
420 POKE 704+X,Z:NEXT X:FOR Y=1 TO 5:NEXT Y:GOTO 400
```

ROLL

```
5 DEG
10 GRAPHICS 10
15 FOR I=0 TO 7:POKE 705+I,128+2:NEXT I
17 POKE 705,136
20 FOR ANG=180 TO 360+180 STEP 6
30 X=8+8*COS(ANG)
40 Y=16+8*SIN(ANG)
50 COLOR (ANG-180)/45+1:PLOT X,Y
60 DRAWTO X,50+Y
70 COLOR 0:PLOT X,Y
90 NEXT ANG
120 FOR ANG=180 TO 360+180 STEP 6
130 X=26+8*COS(ANG)
140 Y=16+8*SIN(ANG)
150 COLOR 9-(ANG-180)/45:PLOT X,Y
160 DRAWTO X,50+Y
170 COLOR 0:PLOT X,Y
190 NEXT ANG
220 FOR ANG=180 TO 360+180 STEP 6
230 X=44+8*COS(ANG)
240 Y=16+8*SIN(ANG)
250 COLOR (ANG-180)/45+1:PLOT X,Y
260 DRAWTO X,50+Y
270 COLOR 0:PLOT X,Y
290 NEXT ANG
320 FOR ANG=180 TO 360+180 STEP 6
330 X=62+8*COS(ANG)
340 Y=16+8*SIN(ANG)
350 COLOR 9-(ANG-180)/45:PLOT X,Y
360 DRAWTO X,50+Y
370 COLOR 0:PLOT X,Y
390 NEXT ANG
410 GO TO 500
420 FOR ANG=180 TO 360+180 STEP 6
430 X=50+8*COS(ANG)
440 Y=16+8*SIN(ANG)
450 COLOR (ANG-180)/45+1:PLOT X,Y
460 DRAWTO X,50+Y
470 COLOR 0:PLOT X,Y
490 NEXT ANG
500 A=PEEK(705)
510 FOR I=705 TO 711
520 POKE I,PEEK(I+1)
530 NEXT I
540 POKE 712,A
550 GO TO 500
```

RING

```
100 REM GTIA TEST
110 DIM C(22,2)
115 GRAPHICS 10:FOR Z=704 TO 712:READ R:POKE Z,R:NEXT Z
116 DATA 0,26,42,58,74,90,106,122,138,154
118 LIM=22:T2=3.14159*2/LIM
120 GOSUB 2500:FOR V=1 TO LIM:T=T+T2:GOSUB 2500:NEXT V
200 GOTO 1000
400 FOR X=1 TO 8:Z=PEEK(704+X):Z=Z+16:IF Z>255 THEN Z=26
420 POKE 704+X,Z:NEXT X:POKE 77,0:GOTO 400
1000 REM
1010 FOR R=1 TO 8:T6=R
1020 GOSUB 1520:NEXT R
1110 FOR R=9 TO 15:T6=16-R
1120 GOSUB 1520:NEXT R
1210 FOR R=16 TO 23:T6=R-15
1220 GOSUB 1520:NEXT R
1310 FOR R=24 TO 30:T6=31-R
1320 GOSUB 1520:NEXT R
1400 IF T3=1 THEN GOTO 400
1410 T3=1:GOTO 1010
1520 COLOR T6:V=0:GOSUB 2000:PLOT X,Y:FOR V=1 TO LIM:T=T+T2:
     GOSUB 2000:GOSUB 3000:DRAWTO X,Y:NEXT V:RETURN
2000 X=(30-R)*C(V,1)+40:Y=(60-R)*C(V,2)+80:RETURN
2500 C(V,1)=SIN(T):C(V,2)=COS(T):RETURN
3000 IF T3=1 THEN IF (R=1 AND V>11) OR R>1 THEN POSITION
     X,Y:POKE 765,T6:XIO 18, #6,0,0,"S:"
3010 RETURN
```

BALL

```
100 REM GTIA TEST
115 DIM C(8):GRAPHICS 10:FOR Z=704 TO 712:READ R:R=R*16+8:C(Z-704)=R:POKE Z,R:NE
XT Z
116 DATA -.5,1,3,4,5,7,9,12,13
118 LIM=22:T2=3.14159*2/LIM:COL=3:E1=1:DIM D(LIM,2)
120 GOSUB 1500:FOR V=1 TO LIM:T=T+T2:GOSUB 1500:NEXT V
400 GOTO 1000
490 REG=705
500 FOR X=1 TO 8:POKE REG,C(X):REG=REG+1:IF REG>712 THEN REG=705
510 NEXT X:REG=REG+1:IF REG>712 THEN REG=705
520 POKE 77,0:GOTO 500
1000 REM
1005 FOR E=1 TO 10:E2=INT(E/2-0.5)
1010 FOR R=E1 TO E1+E2:CR=8-COL:IF CR=0 THEN CR=8
1015 V=0:COLOR CR:GOSUB 2000:PLOT X,Y
1020 FOR V=1 TO LIM:T=T+T2:GOSUB 2000:DRAWTO X,Y:IF V>=LIM/2 THEN COLOR COL
1025 NEXT V:NEXT R:COL=COL+1:IF COL=9 THEN COL=1
1030 E1=E1+INT(E/2+0.5):NEXT E
1200 GOTO 490
1500 D(V,1)=SIN(T):D(V,2)=COS(T):RETURN
2000 X=(30-R)*0.6*D(V,1)+40:Y=60*D(V,2)+80:RETURN
```

Graphics Seven Plus

First came the TRS-80, Model I. It provided character-oriented graphics.

Next came the Apple. It provided both character and line graphics (one or the other).

Now we have the Atari. It provides 14 graphics modes, some character-oriented, some line-oriented.

"Fourteen modes?" you say. "The Basic manual lists nine." Well, that's because Basic only allows you to access nine directly. However, there are others lurking within the machine waiting for a programmer to find them. All are variations on the available modes, some quite useful. One is so useful that this article will be devoted to discussing its use.

All character-line graphics on the Atari ("playfield graphics") are generated by the close co-operation of two chips, Antic and CTIA. Antic fetches data for 3.7 million points per second (320 per line x 192 lines x 60 per second) and feeds it to CTIA which generates the TV picture from that data. To determine what sort of image should be generated (character, line, pixel size, etc.), Antic looks to his program, the display list. This program coexists in memory with all the usual Basic and 6502 programs. Anyway, his program, composed of individual instruction codes, tells him what sort of image to generate.

There are 14 image-generating codes in Antic's program. Now when Basic was designed, for some reason it was decided to allow access to only nine of these codes, rather than the full 14. And in particular, the highest resolution four-color mode was left out. This is "graphics 7+" (also known as "graphics seven-and-a-half.")

We got a great deal of mail from people asking how to use this graphics mode when we documented its existence back in the July 1981 *Creative*. (If you wish to see a tutorial on the Atari for the Basic programmer, go back to the June issue and read the "Outpost" columns to date. Sadly, we can't explain how Antic and such work in each article because the explanation is so long, but we can refer you to previous issues to get a background.)

It takes a bit of work and a fair grasp of what goes on inside the Atari, but the results are well worth it: in the highest four-color mode, we can get double the resolution of graphics 7 using graphics 7+.

Graphics 7, you will recall, gives us 96 vertical x 160 horizontal pixels in four colors. Graphics 8 gives us 192 vertical x 320 horizontal, but only in one color. Graphics 7+ gives us 192 vertical x 160 horizontal in four colors.

This is an extremely useful mode. Graphics 8 has several disadvantages; single dots sometimes become red or blue when white was intended because of "artifacting," and candy-stripes tend to appear on all near-vertical lines. Graphics 7 has pixels the size of 2 x 2 graphics 8 dots, and is too "chunky" for really accurate graphics. Graphics 7+, with double the vertical resolution, brings us close to the limits of most monitors in terms of color resolution, with 2 x 1 graphics 8 dots. No artifacting, no funny stripes, just nice colors in truly high resolution.

I should also mention that the graphics 7+ resolution is equal to the resolution of a player or missile at size x1.

Here at Houston Instruments, where I work, we have a project going to interface a plotter, capable of eight colors, to a digitizer. The image to be plotted must be displayed on the TV. Graphics 7 resolution is unacceptable; the individual pixel is too large for a quality display. But graphics 7+ provides twice the resolution while retaining the four colors of data. (Now, you'd like to know how I plan to get eight colors, right? I must confess to having a few sneaky ideas how to do so, and I promise to document the method should I succeed.) However, for now, four colors at 160 x 192 will do nicely.

A Look at Graphics 7 and 8

Graphics 7+ is midway between 7 and 8, so let's look at 7 and 8 to help understand how to generate 7+.

Graphics 7 is a "four color" mode. This means that for every point on screen, two bits of information are saved in memory. Depending on which of the four numbers possible is saved in those two bits, one of four color registers is selected to display color. (Actual color information is not saved in the display memory; rather, a color register number is saved, with the actual color being stored in the register.) Hence, one byte (eight bits) in graphics 7 display memory, looks like this:

ww xx yy zz

where w, x, y, and z are the information for a given point on screen.

Program 1.

```
10 REM PROGRAM 1 -- DAVE SMALL
20 REM PROGRAM TO GENERATE GR.7
30 REM SAMPLE DISPLAY
40 REM
50 REM 8K BASIC VERSION
60 REM
70 GRAPHICS 7
80 COLOR 1
90 PLOT 1,1
100 DRAWTO 159,1
110 COLOR 2
120 DRAWTO 159,80
130 COLOR 3
140 DRAWTO 1,1
141 FOR Z=1 TO 20
142 COLOR (INT(RND(0)*3)+1)
143 PLOT (INT(RND(0)*159)),
    (INT(RND(0)*80))
144 NEXT Z
150 PRINT "NOTE EACH GRAPHICS
    7 PIXEL"
160 PRINT "USES TWO SCAN LINES."
170 GOTO 170
```

Program 2.

```
10 REM PROGRAM 2 -- DAVE SMALL
20 REM PROGRAM TO GENERATE GR.8
30 REM SAMPLE DISPLAY
40 REM
50 REM 8K BASIC VERSION
60 REM
70 GRAPHICS 8
75 SETCOLOR 2,0,0
80 COLOR 1
90 PLOT 1,1
100 DRAWTO 159,1
120 DRAWTO 159,80
140 DRAWTO 1,1
141 FOR Z=1 TO 20
142 COLOR (INT(RND(0)*3)+1)
143 PLOT (INT(RND(0)*159)),
    (INT(RND(0)*80))
144 NEXT Z
150 PRINT "NOTE EACH GRAPHICS
    8 PIXEL"
160 PRINT "USES ONE SCAN LINE."
170 GOTO 170
```

Program 3.

```
10 REM PROGRAM 3
20 REM
30 REM CONVERT GR.7 TO GR.7+
40 REM DAVE SMALL
50 REM 8K BASIC VERSION
60 REM
70 REM CREATE IMAGE
530 REM *************************
540 REM ** FROM
    CREATIVE COMPUTING..
545 REM ** GENERATES MULTICOLOR
    SPIRAL
550 GRAPHICS 7:DEG :DIM C(3)
555 PRINT "CREATING IMAGE."
590 R=20:COLOR 1:C=1
600 X0=79:Y0=47
610 FOR K=0 TO 3:C(K)=K+1*2:NEXT K
620 FOR K=1 TO 3
630 X=X0+R*COS(360):Y=Y0:PLOT X,Y
640 FOR I=0 TO 5*360 STEP 75
650 X=X0+R*COS(I):Y=Y0+R*SIN(I)
660 DRAWTO X,Y
665 C=C+1:IF C>3 THEN C=1
667 COLOR C
670 NEXT I:R=R+12
680 NEXT K
690 Z8=1
700 PRINT "MODIFYING DL."
1000 REM GR.7 TO GR.7+
1010 START=PEEK(560)+256*PEEK(561)
1020 POKE START+3,14+64:REM LMS
1030 FOR Z=START+6 TO START+6+96
1040 IF PEEK(Z)=13 THEN POKE Z,14
1050 NEXT Z
1059 REM REMOVE THIS STOP
     FOR LOOP..
1060 STOP
1100 REM GR.7+ TO GR.7
1110 FOR Z=START+6+96 TO START+6
     STEP -1
1140 IF PEEK(Z)=14 THEN POKE Z,13
1150 NEXT Z
1155 POKE START+3,13+64:REM LMS
1160 GOTO 1020
```

The memory is mapped starting from the upper lefthand corner of the screen, from the beginning of display memory, across the screen, down one line, and so on. Hence, since we have 96 x 160, or 15,360 points, and four points stored per byte, we use 3840 bytes of data.

When Antic generates graphics 7 he does two scan lines of the same data. Hence, each Antic instruction generates two scan lines, and 96 of these instructions generate 192 lines—the height of the screen.

In graphics 8, we only save one bit of information per point. That bit is used to determine at what intensity a point is plotted, and where the background color and intensity and foreground intensity are stored in color registers. Since only one bit is saved per point, a graphics 8 display memory byte looks like this:

a b c d e f g h

where each letter represents one point. There are 320 x 192 points, 8 to a byte, which comes out to 7680 bytes of data.

Each graphics 8 Antic instruction generates one scan line, so there are 192 of them to a full screen.

Now graphics 7+ has the same vertical resolution as graphics 8—one line per Antic instruction. It also has the same horizontal resolution as graphics 7 (160), and the four colors. Do you begin to see why it is such a useful mode?

Note that different information must be written into display memory to draw a line in a different mode. In particular, in graphics 7 or 7+ two bits must be written for each pixel, whereas in graphics 8 one bit must be written. This will be very important shortly. An operating system routine, stored in the ROM plug-in cartridge, handles all of the bit-shifting and masking to write the required bits into memory, based on what graphics mode it thinks it is in.

Time for some sample programs: The first generates a simple graphics 7 display. The next generates a simple graphics 8 display. This is to allow you to compare the resolutions. See Programs 1 and 2.

Next, we will take a graphics 7 display and convert it to graphics 7+.

What will happen? Well, first, since we have 96 instructions in graphics 7, each generating two scan lines, we get a total of 192 scan lines. If each of those 96 instructions generates only one scan line, as in graphics 7+, the screen will only be half filled (only the top 96 scan lines). The same display that graphics 7 had in it will be retained, it will just shrink vertically.

So for our third program, let's take a graphics 7 display, and convert it to graphics 7+. You'll see the effect of doubling your vertical resolution, and won't believe how fine a line can be drawn in four colors. All we'll do is take the 96 bytes of Antic's program, when he's in graphics 7, and convert them from an Antic code 13 (graphics 7) to a 14 (graphics 7+). See Program 3.

Pretty neat, right? Nice resolution. Now if we could only get the whole screen in that resolution.

Well, we can. We could go the tough way, where we allocate memory, build 192 graphics 7 (14) instructions, set memory pointers to display memory, *ad infinitum*. Were we working in assembly language, we would have to do it that way. But there's an easier way: take an existing display list and convert it. That way Basic has already allocated memory space and so forth, and we don't need to worry about fooling it into leaving memory alone.

We can take a graphics 8 display list, already 192 instructions long, and convert the 15's (Antic code for graphics 8) to 14's. That part is easy, just a FOR-NEXT loop to convert every 15 to a 14. The only slightly tricky part is catching the LMS instructions (64 + 15 or 79), changing them to 78, and leaving the display memory data bytes alone. (See August 1981 for a discussion of LMS). This way, the right amount of screen memory is already reserved for us, the display list is set up, pointers and all, and we've saved a great deal of work.

Next, since graphics 8 uses a different bit pattern to display material, we'll have to fool the operating system into thinking we're really in graphics 7 so it uses the graphics 7 bit/shift routines. This is a matter of one POKE to the low memory location where the operating system looks each time it does a line draw to determine what graphics mode it is in. The location contains the graphics number currently in effect. We will, thus, POKE a 7 in there; it should currently contain an 8 from when graphics 8 was set up.

Well, here we go. (See Program 4.) We set up graphics 8, change the display list to graphics 7+, and do a three-color draw at the top of the screen. No problem, works fine. But when we try to draw anywhere in the lower half of the screen, we get an ERROR #141—cursor out of range.

Many, many people have tried the above routine to get into graphics 7+. All of them have run into this problem. You see, the operating system, while drawing a line, constantly checks to see if the line is going off of the visible area. Should it do so, an ERROR 144 is returned and the line drawing process stops. The OS thinks we're in graphics 7 (96 x 160), so when we try to draw below line 96, it thinks it is at the bottom of the screen and terminates the draw. In computerese this is known as "bounds checking"—and anyone who has watched football knows what "out of bounds" means. (See, these computer snob words really do have humble beginnings).

Program 4.

```
 10 REM PROGRAM 4
 20 REM
 30 REM CONVERT GR.8 TO GR.7+
 40 REM DAVE SMALL
 50 REM 8K BASIC VERSION
 60 REM
 65 DIM C(3)
 70 REM DISPLAY LIST MODS
 80 GRAPHICS 8
 90 PRINT "CONVERTING DL
    FROM 8 TO 7+."
100 START=PEEK(560)+256*PEEK(561)
110 POKE START+3,14+64
120 FOR Z=START+6 TO START+6+192+6
130 IF PEEK(Z)=15 THEN POKE Z,14
140 IF PEEK(Z)=15+64 THEN POKE Z,
    14+64:Z=Z+2:REM
    (SKIP LMS DATA BYTES)
150 NEXT Z
200 REM
210 REM LET OS THINK WERE IN GR.7..
220 POKE 87,7
390 PRINT "CREATING UPPER
    HALF IMAGE"
400 YADD=1
410 GOSUB 500
420 PRINT "CREATING LOWER
    HALF IMAGE"
425 YADD=30
430 GOSUB 500
440 STOP
500 REM
530 REM ***************************
540 REM ** FROM CREATIVE COMPUTING..
545 REM ** GENERATES
    MULTICOLOR SPIRAL
550 DEG
590 R=10:COLOR 1:C=1
600 X0=79:Y0=47
610 FOR K=0 TO 3:C(K)=K+1*2:NEXT K
620 FOR K=1 TO 3
630 X=X0+R*COS(360):Y=Y0:PLOT
    X,Y+YADD
640 FOR I=0 TO 5*360 STEP 75
650 X=X0+R*COS(I):Y=Y0+R*SIN(I)
660 DRAWTO X,Y+YADD
665 C=C+1:IF C>3 THEN C=1
667 COLOR C
670 NEXT I:R=R+12
680 NEXT K
690 Z8=1
700 RETURN
```

What Do We Do?

We can't POKE an 8 into the OS location, because then the draw routine will use the wrong bit shifting routine and we'll get all sorts of crazy bit patterns and colors. (Feel free to try it—there are many interesting effects obtainable this way. Just delete the POKE 87,7 in Program 3.) And we can't get by with a POKE 7...because then the OS thinks we're going out of bounds. Because both bounds checks and draw routine selection are based on the same location, we're stuck. (The memory location is called DINDEX and is located at 57 hex or 87 decimal).

The problem resides in the extreme care

taken to avoid out-of-bounds conditions. If we could draw out of bounds, and have the Atari blindly do the draw instead of telling us we were wrong, then graphics 7+ would work. Even though the operating system might conclude that we were out of our minds and drawing off the bottom edge of the screen, it would continue to draw in the right places for our graphics 7+ to work. (Screen memory, by the way, is 3780 bytes in graphics 7 and 7680 in graphics 7+. Graphics 7+ and graphics 8 use the same memory size.)

Well, the OS routine is in ROM and cannot be modified, short of pulling the chips out and putting new ones in. As I am no hardware expert this solution isn't acceptable. Besides, if I did, my programs would run only on my machine. However, it did bring to mind an analogy which solved the problem. Character sets are stored in ROM, also, and are unmodifiable, unless they are copied into RAM first. So why not copy the OS draw routine into RAM, zap the bounds check, and use it for graphics 7+?

To make a long story even longer, that's what I did. The rest of the article describes this process. The first time through, I did it all in Basic, but that was too slow, so I recoded the slow parts in 6502 assembler. Those routines I used in the graphics 7+ driver. (They should be usable in any graphics mode; they just ignore all bounds checks. However, the Atari caution extends beyond overprotecting the user; a line drawn out of bounds could go sailing straight through memory reserved for other things, and crash the Atari. Just be careful; don't try to draw from 1,1 to 3000,6700.)

The final result is three assembly routines. They are fast and efficient and both fit into page 6 in memory (600-700 hex), 256 bytes set off by Atari for a user's own purposes and left untouched by Atari routines. The first modifies the graphics 8 display list to a graphics 7+. The second copies the OS draw routine into free RAM for modification. I use Basic for the small amount of POKEing that must be done in the OS routine to make it work properly in its new memory location (it involves relocating a few addresses) and to DRAW a line using the OS routine (it just takes arguments from the Basic USR call and feeds them to the draw routine).

To use graphics 7+, one does a graphics 8 call, calls the first USR routine to set up the 7+ display list, calls the second routine to fetch the draw routine in RAM and modify it, and then all is ready. Line draws are made in one of two forms:

X=USR(third routine,X coordinate, Y coordinate, color #) or X=USR(third,X1, Y1,X2,Y2,COLOR)

Program 5.

```
0            0250          *=    $0680
02FE           1390 ATACHR =     $2FE         COLOR DATA
05A          1340 OLDROW =     $5A          FROM Y
06FD 4CFC7C 1790           JMP   $7CFC        O.S...MUST MOD
10 ;
20 ; PROGRAM 5 LISTING..
22          1400 ICCOMZ =     $22          CIO DRAW FLAG
30 ;
40 ; THREE ASSEMBLY ROUTINES FOR
50 ; PAGE 6:
54          1370 ROWCRS =     $54          TO Y
55          1360 COLCRSL=     $55          TO X LO
56          1350 COLCRSH=     $56          TO X HI
57          1380 DINDEX =     $57          CURR GR. MODE
60 855C    1500          STA   OLDCOLH
68D 8D9706 0330          STA   FETCHH        (FETCH STMT)
69E D005   0520          BNE   NOT15
70 ; 1.CONVERTS DL FROM GR.8 - GR7.5.
80 ; 2.COPIER FROM OS ROM TO RAM.
90 ; 3.GR7.+ DRAWTO. FULL SCREEN
0100 ;    GR.7+ DRAW ROUTINE.
0110 ;
0120 ; COPYRIGHT 1981 BY DAVID M. SMALL
0130 ;
0140 ;-----------------------------
0150 ; ROUTINE 1:
0160 ; ASSEMBLY ROUTINE TO CONVERT
0170 ; A GR.8 DISPLAY LIST TO A GR 7.+
0180 ; DISPLAY LIST.
0190 ; CONVERTS ALL 15'S TO 14'S
0200 ; CONVERTS ALL (64+15) TO (64+14)
0210 ; (BUT WILL SKIP LMS DATA BYTES)
0220 ;
0230 ; PLACED IN PAGE 6.
0240 ;
0270 ;
0310 ;
0350 ;
0370 ;
0380 ; LOOP 202 TIMES. CHANGE 15 TO
0390 ; 15, 79 TO 78, SKIP LMS DATA.
0400 ;
0440 ;
0450 ; IF GR.2 ENCOUNTERED, QUIT --
0460 ; HAS A TEXT WINDOW.
0470 ;
0500 ;
0550 ;
0590 ;
0660 ;
0680 68     0260          PLA                    SATISFY BASIC
0681 AD3002 0280          LDA   560
0684 8D9606 0290          STA   FETCHL        (FETCH STMT)
0687 8DAC06 0300          STA   STOREL        (STORE STMT)
0690 ;
0693 A200   0360          LDX   #0             INIT X
0695 BD3412 0430 LOOP     LDA   $1234,X     GET DL BYTE
0696        0410 FETCHL =      *+1
0697        0420 FETCHH =      *+2
0698 C942   0480          CMP   #66
0720 ;
0740 ;
0750 ;-----------------------------
0760 ; ROUTINE 2:
0770 ;
0780 ; COPIES O.S. ROM TO RAM (DRAW
0790 ; ROUTINES) TO ALLOW BOUNDS
0800 ; CHECK REMOVAL.
0810 ;
0820 ; COPIES $FCFC TO $FE44
0830 ;     TO $7CFC TO $7E44
0840 ;
0850 ; (THIS IS QUITE EASY TO CHANGE
0860 ;  TO CUSTOMIZE FOR YOUR ATARI;
0870 ;  ON A 40K-48K MACHINE THIS
0880 ;  IS RIGHT BELOW THE DL/DM.)
0890 ;
0900 ; (65092-64764=  328
0910 ; 328 - 256  = 72 )
0920 ;
0930 ;-- $FCFC TO $FDFB ($FF BYTES)
1010 ;
1020 ;-- $FDFC TO $FE44
1100 ;
```

The first performs a DRAWTO from the old cursor location to the specified X and Y coordinates. The second performs a line draw between the specified points (equivalent to PLOT X1,Y1, : DRAWTO X2,Y2). Both routines perform the draw in the specified color, not the color of the current COLOR statement.

Alas, the OS draw routine is too long to fit into the small page 6. So it must be stored elsewhere in RAM. Finding a free space in RAM isn't too hard. However, finding a space that is free on *everyone's* Atari is pretty hard. Memory sizes range from 8K to 48K (40K with Basic cartridge). I decided to tailor the routine for my 40K system and let users do relocation as necessary for their own systems. Nowadays there is so much player-missile memory being reserved, charset arrays, and so forth that a general solution is very difficult.

For Advanced Programmers

The following is a bit technical but is intended for assembly programmers. The OS routines start at $FCFC and end at $FE44 (inclusive). They are copied to $7CFC through $7E44. Several JMPs inside are relocated back to the RAM routine, making this a non-relocatable routine. (The fact that I am copying it down an even $8000 makes it quite easy to relocate.) It should be simple to do this for other size memories; the calculations are self-documenting in the OS and assembly listings. Just make sure the JMPs are changed to JMP to the point in RAM where the corresponding statement to the ROM statement is. Note that $7E44 is just below the DL/DM in a 40K or 48K (same thing with a Basic cartridge) machine. Hence it is in a relatively "safe" area.

Program 5, continued

```
1110 ; ----------------------------
1120 ; ROUTINE 3:
1130 ;
1140 ; THIS ROUTINE IS CALLED FROM
1150 ; BASIC TO PERFORM A DRAWTO
1160 ; FUNCTION IN GR 7.5. THERE ARE
1170 ; TWO POSSIBLE CALLS:
1180 ;
1190 ; D=USR(X1,Y1,X2,Y2,COLOR)
1200 ; D=USR(X2,Y2,COLOR)
1210 ;
1220 ; FIRST WILL DRAW A LINE BETWEEN
1230 ; THE SPECIFIED COORDINATES IN
1240 ; SPECIFIEDCOLOR. SECOND WILL
1250 ; "DRAWTO" FROM OLD LOCATION TO
1260 ; SPECIFIED COORDINATES.
1270 ;
1280 ; THIS ROUTINE REQUIRES THE O.S.
1290 ; DRAW ROUTINE BE COPIED INTO
1300 ; RAM AND MODIFIED. SEE ARTICLE.
1310 ;
1410 ; PULL OFF AND STORE ARGS
1480 ;
1560 ;
1610 ;
1650 ;
1690 ;
1700 ; SETUP IS DONE. OTHER MISC:
1710 ;
1760 ;
1770 ; CALL DRAW RAM ROUTINE
1780 ;
1800 END
6000 68      1510          PLA                 GET FROM X LO
060000 855B    1520            STA  OLDCOLL
```

Program 6.

```
9000 REM LOADER
9010 Z=6*256+8*16
9020 READ Z1
9030 IF Z1=-1 THEN RETURN
9040 POKE Z,Z1
9050 Z=Z+1
9060 GOTO 9020
10000 DATA 104,173,48,2,141,150,6,141,172,6,173,49,2,
      141,151
10010 DATA 6,141,173,6,162,0,189,52,18,201,66,240,29,
      201,15
10020 DATA 208,5,169,14,76,171,6,201,79,208,2,169,78,
      157,52
10030 DATA 18,232,201,79,208,2,232,232,224,203,144,220,
      96,162,0
10040 DATA 104,189,252,252,157,252,124,232,224,0,208,
      245,162,0,189
10050 DATA 252,253,157,252,125,232,224,75,208,245,96,
      104,201,3,240
10060 DATA 15,201,5,240,1,96,104,133,92,104,133,91,
      104,104,133
10070 DATA 90,104,133,86,104,133,85,104,104,133,84,
      104,104.141,251
10080 DATA 2,169,17,133,34,76,252,124
11000 DATA -1
```

Program 7.

```
10 REM PROGRAM   -- ASSEMBLY VERSION
15 REM REQUIRES AUTORUN.SYS OR LOAD
20 REM
40 REM DAVE SMALL
50 REM 8K BASIC VERSION
55 REM
56 IF PEEK(1536+128)<>104 THEN PRINT "ASSEMBLY
   NOT LOADED..":STOP
60 REM DEFINES
61 CONVERT=6*256+8*16:REM $0680
62 COPY=6*256+11*16+10:REM $06BA
63 DRAW=6*256+13*16+6:REM $06D6
65 DIM C(3)
67 REM
70 REM DISPLAY LIST MODS
80 GRAPHICS 8
90 X=USR(CONVERT)
97 REM
200 REM
210 REM LET OS THINK WE'RE IN GR.7..
220 POKE 87,7
230 REM
300 PRINT "PERFORMING OS COPY."
310 X=USR(COPY)
320 REM RELOCATION
321 POKE (7*4096+13*256+9*16+8),(7*16+14):REM FD98,
    FE TO 7E
322 POKE (7*4096+14*256+2*16+6),(7*16+14):REM FE26,
    FE TO 7E
323 POKE (7*4096+14*256+4*16+1),(7*16+13):REM FE41,
    FD TO 7D
324 REM NOP OUT BOUNDS CHECKS
325 L=7*4096+13*256+15*16+6
326 FOR Z=L TO L+2
327 POKE Z,234:REM NOP
328 NEXT Z
350 REM
390 PRINT "CREATING FULLSCREEN IMAGE"
500 REM
530 REM *************************
540 REM ** FROM CREATIVE COMPUTING..
545 REM ** GENERATES MULTICOLOR SPIRAL
550 DEG
590 R=20:COLOR 1:C=1
600 X0=79:Y0=85
610 FOR K=0 TO 3:C(K)=K+1*2:NEXT K
620 FOR K=1 TO 3
630 X=X0+R*COS(360):Y=Y0+R*SIN(360)
636 Z=USR(DRAW,X,Y,X,Y,0):REM (PLOT)
640 FOR I=0 TO 5*360 STEP 75
650 X=X0+R*COS(I):Y=Y0+R*SIN(I)
662 Z=USR(DRAW,X,Y,C):REM (DRAWTO)
665 C=C+1:IF C>3 THEN C=1
670 NEXT I:R=R+20
680 NEXT K
690 Z8=1
700 STOP
```

The bounds check is a simple JSR. This is changed to NOP (no-operation) with three NOP codes.

Programs 5, 6, 7, and 8 are listings of four assembly/Basic routines. (The Atari OS listing is copyrighted and doesn't appear here, but you can easily look up the addresses specified to find where I am copying from yourself.)

Program 5 is the page 6 assembly listing. Program 6 is the assembly program converted to DATA statements. This program is appended to your code to load the assembly routine. Program 7 is the "Sunset" multiple color spiral run in graphics 7+, using an already loaded assembly routine, and provides an example of using graphics 7+ when the routines are loaded. Finally, Program 8 is an example of using the DATA statements of Program 6 to load and draw a pretty figure using graphics 7+.

Feel free to delete the REM statements; I document the code heavily in order to make it easy to understand, but the documentation isn't needed in the final copy. (I also break up all hex opcodes for clarity; these could be calculated to save the machine the work each runthrough.)

On using AUTORUN.SYS: This is a handy way for disk users to load these routines. Boot up DOS (2.0S), and run Program 6. Next, go to DOS. Do the binary save (K), from $600 to $6FF:

K

AUTORUN.SYS,600,6FF (return)

and thereafter when you boot up with that disk, the graphics 7+, routines will be loaded automatically.

Generally DOS and Basic will leave these routines alone once loaded unless you reboot the system or have a particularly nasty crash. Hence, even users without disks may not have to reload the data each program run.

Conclusion

Well, there you have it, graphics 7+. I hope to see more and more use of it! These routines can easily be copied into a AUTORUN.SYS file and automatically loaded along with Basic, or POKEd into memory when needed. Enjoy the world of double resolution graphics 7. □

Program 8.

```
10 REM PROGRAM 8 -- DEMOS LOAD THRU
15 REM DATA STATEMENTS.
20 REM
40 REM DAVE SMALL
50 REM 8K BASIC VERSION
54 GOSUB 9000
55 REM
56 IF PEEK(1536+128)<>104 THEN PRINT
   "ASSEMBLY NOT LOADED..":STOP
60 REM DEFINES
61 CONVERT=6*256+8*16:REM $0680
62 COPY=6*256+11*16+10:REM $06BA
63 DRAW=6*256+13*16+6:REM $06D6
65 DIM C(3)
67 REM
70 REM DISPLAY LIST MODS
80 GRAPHICS 8+16
95 X=USR(CONVERT)
96 GOTO 200
97 REM
200 REM
210 REM LET OS THINK WE'RE IN
    GR.7..
220 POKE 87,7
230 REM
300 REM
310 X=USR(COPY)
320 REM RELOCATION
321 POKE (7*4096+13*256+9*16+8),
    (7*16+14):REM FD98, FE TO 7E
322 POKE (7*4096+14*256+2*16+6),
    (7*16+14):REM FE26, FE TO 7E
323 POKE (7*4096+14*256+4*16+1),
    (7*16+13):REM FE41, FD TO 7D
324 REM NOP OUT BOUNDS CHECKS
325 L=7*4096+13*256+15*16+6
326 FOR Z=L TO L+2
327 POKE Z,234:REM NOP
328 NEXT Z
350 REM
390 REM
400 SETCOLOR 0,2,4:REM RED
410 SETCOLOR 1,7,4:REM BLUE
420 SETCOLOR 2,13,4:REM GREEN
500 DEG
505 X2=SIN(0)*70+70:Y2=COS(0)
    *80+80
507 Z=USR(DRAW,X2,Y2,X2,Y2,0)
    :REM PLOT
508 C=1
510 FOR X=0 TO 360 STEP 4
520 X1=SIN(X*1.5)*70+70
530 Y1=COS(X*2)*80+80
531 X2=SIN(X+120)*40+60
532 Y2=COS(X-40)*50+60
540 Z=USR(DRAW,X1,Y1,X2,Y2,C)
545 C=C+1:IF C=4 THEN C=1
550 NEXT X
560 GOTO 560
9000 REM LOADER
9010 Z=6*256+8*16
9020 READ Z1
9030 IF Z1=-1 THEN RETURN
9040 POKE Z,Z1
9050 Z=Z+1
9060 GOTO 9020
9999 REM DATA FOR GR 7+ DRIVER
10000 DATA 104,173,48,2,141,150,6,
      141,172,6,173,49,2,141,151
10010 DATA 6,141,173,6,162,0,189,
      52,18,201,66,240,29,201,15
10020 DATA 208,5,169,14,76,171,6,
      201,79,208,2,169,78,157,52
10030 DATA 18,232,201,79,208,2,
      232,232,224,203,144,220,96,
      162,0
10040 DATA 104,189,252,252,157,
      252,124,232,224,0,208,245,
      162,0,189
10050 DATA 252,253,157,252,125,
      232;224,75,208,245,96,104,
      201,3,240
10060 DATA 15,201,5,240,1,96,104,
      133,92,104,133,91,104,
      104,133
10070 DATA 90,104,133,86,104,133,
      85,104,104,133,84,104,104,
      141,251
10080 DATA 2,169,17,133,34,76,
      252,124
11000 DATA -1
```

Player-Missile Design Aid

Tom Gurak

Player/Missile Design Aid (PMDA) is a program which aids you in designing your own player/missile graphics. Player/missile graphics are a powerful tool provided by Atari for designing games. However, designing and encoding each player/missile character can be a time-consuming process. Further, using the normal method of designing these players on graph paper, the designer is never sure exactly how the player/missile graphic will look when displayed on the screen.

Player/Missile Design Aid was written to facilitate this process and allow the designer to see the player/missile graphic he is designing while he is working on it.

Whenever PMDA is awaiting your direction, it shows a blinking cursor on the screen. To move the cursor, simply push the joystick in the direction you wish to move the cursor. The cursor will continue to move in that direction until you release the joystick or push it in a different direction.

To start, LOAD the PMDA program and type RUN. PMDA will then display a title screen and begin setting up. Once setup is complete, PMDA displays a screen containing an 8 x 24 bit map which will be used to design your player graphic.

Note that a bit which is off (0) is displayed as a plus sign (+) and a bit which is on (1) is displayed as a solid white block. To the immediate left of the bit map is a column of line numbers and to the right is the decimal POKE value for each line. Initially, this latter field is all zeroes. As bits are turned on, however, this will change to correspond to the new value of the line (byte).

On the right side of the screen is a list of commands, a status line, and a prompt line which indicates the action to be taken.

Some explanation of the status line is in order. The first item is the current player/missile mode (M=nn). The two digits are the actual decimal value which is POKEd at SDMCTL (559) to produce the desired mode. M=46 indicates that you are in double-line mode (the default); M=62 indicates that you are in single-line mode.

The second item is the player size or width (W=n). The digit following is the desired value to be POKEd in the player size register (in this case, SIZEP0 (53256)). W=0 indicates single width (the default); W=1 indicates double width; and W=3 indicates quadruple width. The last item is the color/luminance for the player/missile graphic (COLOR=). The digits following are the actual decimal POKE value in the player/missile color register (in this case, PCOLR0 (704)).

I would like to point out that I am not attempting to explain player/missile graphics as there has been much information published already on this subject. I am merely attempting to present enough information to enable you to understand the operation of the Player/Missile Design Aid.

Finally, we are ready to begin designing our player/missile graphic. Using the joystick, position the blinking cursor to the bit position in the map which is to be changed. Pushing the fire button on the joystick will cause the bit to be flipped from off to on, or vice-versa. As bits are turned on, the actual player/missile graphic will begin to take shape in the area between the bit map display and the command list.

It is also possible to "draw" a line in any direction. To accomplish this, position the cursor to the desired starting position of the line, press and *hold* the fire button, and push the joystick in the desired direction. Remember that if you pass over a bit position which is already on, it will be turned off.

To use the commands (each of which is described later), position the cursor to the first character of the desired command and press the fire button. The command list may be reached by moving the cursor to the left or right until it leaves the bit map display. To return the cursor to the bit map, simply move the joystick left or right.

When the player/missile graphic is completed and all options (mode, width, and color) are set correctly, you can either write down the status line settings and the decimal values for each line (byte) of the player/missile graphic or you can use the Save Data command to save this data. The data saved takes the form of a Basic language DATA statement which may be added to your own player/missile graphic program by using the Atari ENTER command. This eliminates the need for a runtime subroutine to load the data. The format of the DATA statement is explained later.

Commands

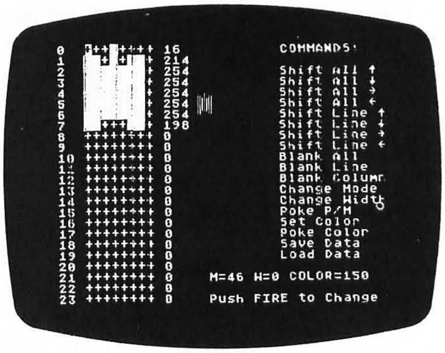
Shift All ↑ : Shifts all 24 lines of the graphic up one line and leaves a blank (0) line at line 23.

Shift All ↓ : Shifts all 24 lines of the graphic down one line and leaves a blank (0) line at line 0.

Shift All →: Shifts all 24 lines of the graphic right one bit position and leaves a blank (0) column of bit positions at the extreme left.

Shift All ←: Shifts all 24 lines of the graphic left one bit position and leaves a blank (0) column of bit positions at the extreme right.

Shift Line ↑ : Shifts all lines from the line you indicate to line 23 up one line and leaves a blank line (0) at line 23. Select the first line to be shifted by positioning the cursor on the desired line and pressing the fire button when prompted by the program.



```
0  +++█++++ 16       COMMANDS:
1  ██+█+██+ 214
2  ███████+ 254      Shift All ↑
3  ███████+ 254      Shift All ↓
4  ███████+ 254      Shift All →
5  ███████+ 254      Shift All ←
6  ███████+ 254      Shift Line ↑
7  ██+++██+ 198      Shift Line ↓
8  ++++++++ 0        Shift Line →
9  ++++++++ 0        Shift Line ←
10 ++++++++ 0        Blank All
11 ++++++++ 0        Blank Line
12 ++++++++ 0        Blank Column
13 ++++++++ 0        Change Mode
14 ++++++++ 0        Change Width
15 ++++++++ 0        Poke P/M
16 ++++++++ 0        Set Color
17 ++++++++ 0        Poke Color
18 ++++++++ 0        Save Data
19 ++++++++ 0        Load Data
20 ++++++++ 0
21 ++++++++ 0    M=46 W=0 COLOR=150
22 ++++++++ 0
23 ++++++++ 0    Push FIRE to Change
```

Tank.

Shift Line ↓ : Shifts all lines from the line you indicate to line 23 down one line and leaves a blank (0) line at the line selected. Line selection is the same as described for Shift Line ↑ above.

Shift Line →: The single line which you select is shifted right one bit position and a 0 bit is left at the extreme left of the line. Line selection is the same as for Shift Line ↑.

Shift Line ←: The single line which you select is shifted left one bit position and a 0 bit is left at the extreme right of the line. Line selection is the same as for Shift Line ↑.

Blank All: All bit positions are set to 0. Before proceeding, you will be asked to confirm your request by pressing the fire button. If you do not want the command to proceed, push the joystick in any direction.

Blank Line: The single line which you select will have all its bit positions set to 0. Line selection is as described for Shift Line ↑.

Blank Column: The bit position which you select will be set to 0 in all lines. Select the bit position by moving the cursor to the desired position and pressing the fire button when prompted by the program.

Change Mode: This changes the mode from double-line (M=46) to single-line (M=62) and vice-versa.

Change Width: This changes the player/missile graphic width from single (W=0) to double (W=1); double to quadruple (W=3); or quadruple to single.

POKE P/M: This allows the user to enter a previously-defined character when only the POKE values are known. Use the

Tom Gurak, 24 North St., W. Albany, NY 12205.

keyboard to enter the value for each line when prompted by the program. The Return key must be pressed after each value. Enter three nines (999) followed by Return to indicate that you are done.

Set Color: This sets the color of the player/missile graphic *only*. Using the keyboard, enter the Atari color value (0-15) followed by Return, then enter the luminance value (0-14, even numbers only) also followed by Return. These values will be converted to the corresponding color register value and POKEd into PCOLR0 to change the color of the player/missile graphic displayed.

POKE Color: This sets the color of the player/missile graphic *only*. Using the keyboard, enter the decimal value to be POKEd into the player/missile color register.

Save Data: This saves the player/missile data as a Basic language DATA statement. The format on this statement is described later. Prior to beginning the operation, you are asked to confirm your intent by pushing the fire button. To cancel the operation, push the joystick in any direction. The data saved include the mode, width, and color settings followed by the POKE values for each line from 0 to the last non-zero line.

Load Data: This loads previously-saved player/missile data. Before beginning the operation, you are asked to confirm your intent by pressing the fire button. To cancel the operation, push the joystick in any direction. Upon confirmation, a Blank All operation will be performed. The player/missile graphic will be loaded and displayed with the same mode, width, and color as were in effect when it was saved.

Messages

Color?: Use the keyboard to enter the Atari color value and press the Return key.

Enter POKE Values: Use the keyboard to enter the POKE values for a play/missile graphic. Press Return after each one and use 999 followed by Return to indicate you are finished.

Luminance?: Use the keyboard to enter the Atari luminance value and press Return.

No P/M Data to Save: The Save Data command was selected but there are no non-zero bits in the bit map. No action is required.

POKE Color?: Use the keyboard to enter the POKE value for the player/missile color register and press the Return key.

Pos Cursor for Blank: Position the cursor to the line/column to be blanked and press the fire button to complete the Blank command.

Pos Cursor for Shift: Position the cursor to the appropriate line for the Shift operation and press the fire button to complete the Shift command.

Processing...: A long-running command is executing. No action is required.

Push FIRE to Change: The cursor is located within the bit map and pressing the fire button will cause the bit at the cursor position to be flipped.

Push FIRE to Confirm: A Blank All, Save Data, or Load Data command has been selected and pressing the fire button will cause the command to continue. The command may be cancelled by pushing the joystick in any direction.

Push FIRE to Select: The cursor is located within the command list and pressing the fire button will cause the command at which the cursor is positioned to be executed.

Ready Tape Recorder: Insert a cassette tape, press Play or Record and Play depending on the operation selected, and press the console Return key.

Save Data Format

The Save Data command produces a Basic language DATA statement which has the following format:

Lineno DATA mode, width, color, data0, data1,...datan,-1

Lineno is the line number. The first save will create a statement with a line number of 32000. For each subsequent save, the line number is incremented by 10.

DATA is written as shown to identify the Basic language statement type.

Mode is the POKE value for the player/missile mode (double-line or single-line).

Width is the POKE value for the player/missile size register.

Color is the POKE value for the player/missile color register.

Data0 is the POKE value needed to create line 0 of the player/missile graphic.

Data1 is the POKE value needed to create line 1 of the player/missile graphic.

Datan is the POKE value needed to create line n of the player/missile graphic. The last line saved is the last non-zero line found in the bit map. Leading zero lines and any zero lines within the body of the player/missile graphic *will* be saved.

-1 is written as shown to indicate the end of the player/missile data. □

```
5 TRAM=PEEK(106)-8:POKE 106,TRAM
10 GRAPHICS 2+16:SETCOLOR 4,9,4:? #6:? #
6:? #6;"   PLAYER/MISSILE":? #6
20 ? #6;"     DESIGN AID":? #6:? #6:? #6
;"        -BY-":? #6:? #6;"       TOM  GUR
AK"
30 K0=0:K1=1:K2=2:K5=5:K7=7:K8=8:K10=10:
K12=12:K13=13:K15=15:K19=19:K22=22:K23=2
3:K27=27:K256=256:K512=512
40 ATRACT=77:SDMCTL=559:PCOLR0=704:CRSIN
H=752:HPOSP0=53248:SIZEP0=53256:GRACTL=5
3277:PMADR=54279
70 PMBASE=TRAM*K256+K512
80 FOR Y=PMBASE TO PMBASE+768:POKE Y,K0:
NEXT Y
90 POKE PMADR,TRAM:PMBASE=PMBASE+34
100 DIM B$(1),I$(1),S$(6),A$(4),L$(5),C$
(6):B$="+":I$="Inverse Video Blank":S$="
Shift ":A$="All ":L$="Line ":C$="Blank "
110 SN=31990:WD=0:MS=46
120 DIM P$(13),Q$(15),T$(7):P$="Push FIR
E to ":Q$="Pos Cursor for "
140 FOR W=K0 TO K256*K10:NEXT W
150 GRAPHICS K0:SETCOLOR K2,K0,K0:SETCOL
OR K1,K12,K12:POKE CRSINH,K1:? " ";
160 POKE SDMCTL,MS:POKE PCOLR0,K12:POKE
GRACTL,3:POKE HPOSP0,119
170 GOSUB 1000
200 POKE ATRACT,K0:LOCATE X+K5,Y,OC:H=12
8:CC=OC+H
210 POSITION X+K5,Y:? CHR$(CC);:H=-H:CC=
CC+H:FOR W=K0 TO K23:NEXT W:P=STICK(K0):
T=STRIG(K0)
215 IF P=K15 AND T THEN 210
220 POSITION X+K5,Y:? CHR$(OC);:IF T THE
N 300
222 IF CSW THEN GOTO CRT
225 IF X=K22 THEN 400
230 CC=ASC(B$):IF OC=CC THEN CC=ASC(I$)
240 POSITION X+K5,Y:? CHR$(CC);:A=PMBASE
+Y:PM=PEEK(A):WM=INT(K2^(K7-X)+0.5):IF O
C=ASC(B$) THEN PM=PM+WM:GOTO 260
250 PM=PM-WM
260 WY=Y:GOSUB 800
270 IF P<>K15 THEN 300
280 P=STICK(K0):IF  NOT STRIG(K0) THEN 2
70
300 XC=K0:YC=K0:IF P>K8 AND P<K12 THEN X
C=-K1:GOTO 320
310 IF P>4 AND P<K8 THEN XC=K1
320 IF P=6 OR P=K10 OR P=14 THEN YC=-K1:
GOTO 335
330 IF P=K5 OR P=9 OR P=K13 THEN YC=K1
335 IF X=K22 AND XC AND YC THEN YC=K0
340 X=X+XC:Y=Y+YC
343 IF CSW THEN GOTO CRT
345 IF X<K8 AND X>=K0 THEN 365
350 IF X=K22 THEN 380
355 IF X=21 OR X=K23 THEN X=K0:Y=K0:GOSU
B 1100:GOTO 200
357 IF  NOT STRIG(K0) THEN 357
360 X=K22:Y=K2:GOSUB 1150:GOTO 200
365 IF Y>K23 THEN Y=K0:GOTO 200
370 IF Y<K0 THEN Y=K23
375 GOTO 200
380 IF  NOT STRIG(K0) THEN 380
385 IF Y<K2 THEN Y=K19:GOTO 200
390 IF Y>K19 THEN Y=K2
395 GOTO 200
400 A=Y-K1:ON A GOTO 410,420,430,440,450
,460,470,480,490,500,510,2300,2200,1500,
900,1600,1700,1900
410 GOSUB 1200:YS=K0:YE=K23:YI=K1:GOTO 7
00
420 GOSUB 1200:YS=K23:YE=K0:YI=-K1:GOTO
700
430 GOSUB 1200:YS=K0:YE=K23:XS=K7:XE=K0:
XI=-K1:GOTO 750
440 GOSUB 1200:YS=K0:YE=K23:XS=K0:XE=K7:
XI=K1:GOTO 750
450 YI=K1:GOSUB 580:YS=YE-K1:YE=K23:GOTO
 700
460 YS=K23:YI=-K1:GOSUB 580:GOTO 700
470 XS=K7:XE=K0:XI=-K1:GOSUB 580:YS=YE:G
OTO 750
480 XS=K0:XE=K7:XI=K1:GOSUB 580:YS=YE:GO
TO 750
490 GOSUB 2500:GOSUB 1200:GOSUB 1000:GOT
O 200
500 GOSUB 590:GOTO 650
510 X=K0:Y=K0:T$=C$(K1,K5):GOSUB 1160:CS
```

```
W=K1:CRT=515:GOTO 200
515 Y=K0:IF  NOT T THEN 530
520 IF X>K7 THEN X=K0:GOTO 200
525 IF X<K0 THEN X=K7
527 GOTO 200
530 GOSUB 1200:WM=INT(K2^(K7-X)+0.5):FOR
 WY=K0 TO K23:LOCATE X+K5,WY,OC:IF OC=AS
C(B$) THEN 550
540 A=PMBASE+WY:PM=PEEK(A):PM=PM-WM:GOSU
B 800
550 POSITION X+K5,WY:? B$;:NEXT WY
570 CSW=K0:IF X=K22 THEN GOSUB 1150:GOTO
 200
575 GOSUB 1100:GOTO 200
580 T$=S$(K1,K5):GOTO 600
590 T$=C$(K1,K5)
600 X=K0:Y=K0:POSITION K19,K23:? Q$;T$;:
GOSUB 1300:CSW=K1:CRT=610:GOTO 200
610 X=K0:IF T THEN 365
620 YE=Y:GOSUB 1200:RETURN
650 POSITION K5,YE:FOR WX=K0 TO K7:? B$;
:NEXT WX
660 A=PMBASE+YE:WY=YE:PM=K0:GOSUB 800:GO
TO 570
700 FOR WY=YS TO YE-YI STEP YI:FOR WX=K0
 TO K7:LOCATE WX+K5,WY+YI,OC:POSITION WX
+K5,WY+YI:? CHR$(OC);
710 POSITION WX+K5,WY:? CHR$(OC);:NEXT W
X:A=PMBASE+WY:PM=PEEK(A+YI):GOSUB 800:NE
XT WY
720 GOTO 650
750 FOR WY=YS TO YE:FOR WX=XS TO XE-XI S
TEP XI
760 LOCATE WX+K5+XI,WY,OC:POSITION WX+K5
+XI,WY:? CHR$(OC);:POSITION WX+K5,WY:? C
HR$(OC);:NEXT WX
770 POSITION XE+K5,WY:? B$;:A=PMBASE+WY:
PM=PEEK(A):IF XI=-K1 THEN PM=INT(PM/K2):
GOTO 780
775 PM=PM*K2:IF PM>=K256 THEN PM=PM-K256
780 GOSUB 800:NEXT WY:GOTO 570
800 POKE A,PM:POSITION 14,WY:? PM;"  ";:
RETURN
900 GOSUB 990:? "Color";:GOSUB 1400:IF P
<K0 OR P>K15 OR P<>INT(P) THEN 900
910 A=P*16
920 GOSUB 990:? "Luminance";:GOSUB 1400:
IF P<K0 OR P>14 OR P<>INT(P/K2)*K2 THEN
920
930 A=A+P:POKE PCOLR0,A
940 GOSUB 950:GOTO 570
950 A=PEEK(PCOLR0):POSITION 28,21:? "COL
OR=";A;"  ";:RETURN
990 POSITION K19,K23:? "
    ";:POSITION K19,K23:RETURN
1000 FOR Y=K0 TO K23:POSITION K2,Y:? Y;:
FOR X=K0 TO K7:POSITION X+K5,Y:? B$;:NEX
T X
1010 A=PMBASE+Y:PM=K0:WY=Y:GOSUB 800:NEX
T Y
1020 POSITION K27,K0:? "COMMANDS:";
1030 POSITION K27,K2:? S$;A$;"Esc Esc Es
c Up-Arrow";:POSITION K27,3:? S$;A$;"Esc
 Esc Esc Down-Arrow";
1040 POSITION K27,4:? S$;A$;"Esc Esc Esc
 Risht-Arrow";:POSITION K27,K5:? S$;A$;"
Esc Esc Esc Left-Arrow";
1050 POSITION K27,6:? S$;L$;"Esc Esc Esc
 Up-Arrow";:POSITION K27,K7:? S$;L$;"Esc
 Esc Esc Down-Arrow";
1060 POSITION K27,K8:? S$;L$;"Esc Esc Es
c Risht-Arrow";:POSITION K27,9:? S$;L$;"
Esc Esc Esc Left-Arrow";
1072 POSITION K27,K13:? "Chanse Mode";
1075 POSITION K27,14:? "Chanse Width";:P
OSITION K27,K15:? "Poke P/M";
1080 POSITION K27,16:? "Set Color";:POSI
TION K27,17:? "Poke Color";
1090 POSITION K27,18:? "Save Data";:POSI
TION K27,19:? "Load Data";:GOSUB 2400:GO
SUB 2250:GOSUB 950
1095 IF CSW THEN RETURN
1100 T$="Chanse ":GOTO 1160
1150 T$="Select ":GOTO 1160
1160 POSITION K19,K23:? P$;T$;:GOSUB 130
0
1170 SOUND K0,K0,K0,K0:RETURN
1200 POSITION K19,K23:? "Processins";:FO
R W=K1 TO K10:? ".";:NEXT W:SOUND K0,250
,6,K2:RETURN
1300 SOUND K0,50,K12,4:FOR W=K0 TO K23:N
EXT W:SOUND K0,K0,K0,K0:RETURN
1400 P=K0:W=K0:OPEN #1,4,0,"K:":GOSUB 13
00:POKE CRSINH,K0:? "?";
1410 GET #1,W:IF W=155 THEN 1490
1420 IF W=126 THEN P=INT(P/K10):? "Esc L
eft-Arrow Space Esc Left-Arrow";:GOTO 14
10
1430 IF W>47 AND W<58 THEN P=P*K10+(W-48
):? CHR$(W);:GOTO 1410
1440 ? "Esc Ctl-Clear";:GOTO 1410
1490 CLOSE #1:POKE CRSINH,K1:? " ";:RETU
RN
1500 POSITION K19,K23:? "Enter Poke Valu
es   ";:FOR WY=K0 TO K23
1510 POSITION K13,WY:GOSUB 1400:IF P=999
 THEN 1590
1520 IF P<K0 OR P>=K256 THEN 1510
1530 PM=P:GOSUB 2100
1560 POSITION K13,WY:? " ";:NEXT WY:GOTO
 570
1590 A=PMBASE+WY:PM=PEEK(A):GOSUB 800:PO
SITION K13,WY:? " ";:POP :GOTO 570
1600 GOSUB 990:POSITION K19,K23:? "Poke
Color";:GOSUB 1400:IF P<K0 OR P>=K256 OR
 P<>INT(P/2)*2 THEN 1600
1610 A=0:GOTO 930
1700 FOR YE=K23 TO K0 STEP -K1:IF PEEK(P
MBASE+YE)<>K0 THEN 1720
1710 NEXT YE:POSITION K19,K23:? "No P/M
Data to Save ";:FOR W=K0 TO K512:NEXT W:
GOTO 570
1720 POP :GOSUB 2500:GOSUB 1790:A=K0
1730 OPEN #1,8,0,"C:":SN=SN+K10:T$=STR$(
SN):GOSUB 1810:T$=" DATA ":GOSUB 1810:T$
=STR$(MS):GOSUB 1810
1735 T$=STR$(WD):GOSUB 1800:T$=STR$(PEEK
(PCOLR0)):GOSUB 1800
1740 FOR WY=K0 TO YE:PM=PEEK(PMBASE+WY):
T$=STR$(PM):GOSUB 1800
1760 NEXT WY:T$="-1":GOSUB 1800:PUT #1,1
55:CLOSE #1:GOTO 570
1790 POSITION K19,K23:? "Reads Tape Reco

rder ";:RETURN
1800 PUT #1,44
1810 FOR WX=K1 TO LEN(T$):PUT #1,ASC(T$(
WX,WX)):NEXT WX:RETURN
1900 GOSUB 2500:GOSUB 1200:CSW=K1:GOSUB
1000:CSW=K0:SOUND 0,0,0,0
1905 GOSUB 1790:OPEN #1,4,0,"C:":FOR WX=
K0 TO K10:GET #1,A:NEXT WX
1910 GOSUB 2000:MS=PM:GOSUB 2400:GOSUB 2
000:WD=PM:GOSUB 2250:GOSUB 2000:POKE PCO
LR0,PM:GOSUB 950
1920 FOR WY=K0 TO K23:GOSUB 2000:IF P=45
 THEN POP :GOTO 1990
1930 P=PM:GOSUB 2100:NEXT WY
1990 CLOSE #1:GOTO 570
2000 PM=K0:FOR WX=K0 TO 4:GET #1,P:IF P=
44 THEN POP :GOTO 2090
2010 IF P=45 THEN POP :GOTO 2090
2020 PM=PM*K10+VAL(CHR$(P)):NEXT WX
2090 RETURN
2100 A=128:FOR WX=K0 TO K7:POSITION WX+K
5,WY:IF P<A THEN 2120
2110 ? I$;:P=P-A:GOTO 2130
2120 ? B$;
2130 A=A/K2:NEXT WX:A=PMBASE+WY:GOSUB 80
0:RETURN
2200 IF WD=K0 THEN WD=K1:GOTO 2230
2210 IF WD=K1 THEN WD=3:GOTO 2230
2220 WD=K0
2230 GOSUB 2250:GOTO 570
2250 POKE SIZEP0,WD:POSITION 24,21:? "W=
";WD:RETURN
2300 GOSUB 1200:IF MS=46 THEN MS=62:GOTO
 2330
2310 MS=46
2330 GOSUB 2400:FOR WY=K0 TO K23:PM=PEEK
(PMSAVE+WY):POKE PMBASE+WY,PM:POKE PMSAV
E+WY,K0:NEXT WY:GOTO 570
2400 POKE SDMCTL,MS:POSITION K19,21:? "M
=";MS;:PMSAVE=PMBASE
2410 WX=K512+34:IF MS=46 THEN 2430
2420 WX=1024+68
2430 PMBASE=TRAM*K256+WX:RETURN
2500 IF  NOT STRIG(K0) THEN 2500
2505 GOSUB 990:POSITION K19,K23:? P$;"Co
nfirm";:GOSUB 1300:LOCATE X+K5,Y,OC:H=12
8:CC=OC+H
2510 POSITION X+K5,Y:? CHR$(CC);:H=-H:CC
=CC+H:FOR W=K0 TO K23:NEXT W:P=STICK(K0)
:T=STRIG(K0)
2520 IF P=K15 AND T THEN 2510
2530 POSITION X+K5,Y:? CHR$(OC):IF P<>K1
5 THEN POP :GOTO 570
2540 RETURN
```

Animath

Jerry Wright and Lloyd Ollman, Jr.

The graphics potential of the Atari personal computer is a powerful educational tool. It can be used to transform the chores of learning into the fun of learning.

A growing number of companies now produce educational software for the Atari computer, but the quality of this software varies widely. A good children's educational program draws children to play with it, and allows learning to happen along with the fun.

When you think back to your school days (assuming you're not still there), what did you find to be the worst part of the learning process? For us it was drill and practice. Here's a children's educational program that makes addition practice enjoyable using an interesting type of animation.

The program is called Animath, for animated math program, and it uses a modified character set to create a sauntering gorilla. Player/missile graphics are also used to spice up the game.

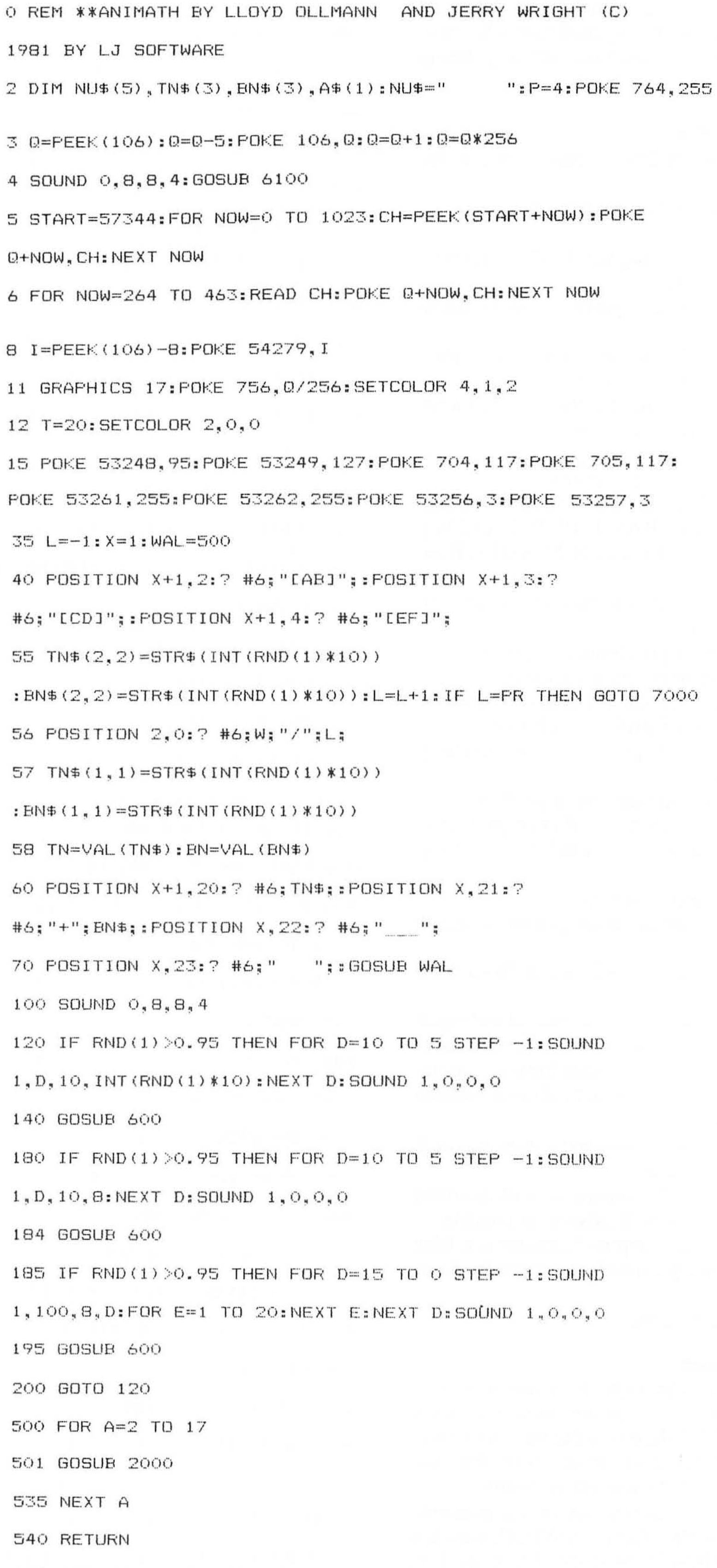
There are several commercial programs which can be used to create modified character sets. Perhaps the best-known of these is *Fontedit*, from Iridis #2. We used a program similar to this to write a "gorilla" font to disk. The original version of this program called the font from disk and loaded it into memory. The Atari character set is a part of ROM, so the font must be moved to RAM, where it can be modified by the appropriate POKEs into memory.

We knew that many Atari owners utilize cassette storage, so we wrote a little utility to save the font as data statements at the end of the program. There are 24 modified characters, represented as 24 data statements. Because the characters are set up in 8 x 8 blocks, each of the eight numbers in the individual data statements is one 8-bit word, or byte.

After the gorilla is POKEd into RAM, he can be animated by the POSITION command. By changing the positions of his arms and legs, we simulate motion, and the gorilla is able to run down the screen to the first problem.

Thanks to Basic A+ from Optimized Systems Software, we were able to get an accurate list of variables. The first list we generated contained several variables we couldn't find. After listing the program to disk and entering it back in the computer, we came up with an accurate variable table, without all the variables that had been eliminated in earlier incarnations of the program. It's always wise to LIST,

Jerry Wright, 18812 116 Ave., SE, Renton, WA 98055.



```
0 REM **ANIMATH BY LLOYD OLLMANN  AND JERRY WRIGHT (C)
1981 BY LJ SOFTWARE
2 DIM NU$(5),TN$(3),BN$(3),A$(1):NU$="     ":P=4:POKE 764,255

3 Q=PEEK(106):Q=Q-5:POKE 106,Q:Q=Q+1:Q=Q*256
4 SOUND 0,8,8,4:GOSUB 6100
5 START=57344:FOR NOW=0 TO 1023:CH=PEEK(START+NOW):POKE
Q+NOW,CH:NEXT NOW
6 FOR NOW=264 TO 463:READ CH:POKE Q+NOW,CH:NEXT NOW

8 I=PEEK(106)-8:POKE 54279,I
11 GRAPHICS 17:POKE 756,Q/256:SETCOLOR 4,1,2
12 T=20:SETCOLOR 2,0,0
15 POKE 53248,95:POKE 53249,127:POKE 704,117:POKE 705,117:
POKE 53261,255:POKE 53262,255:POKE 53256,3:POKE 53257,3
 35 L=-1:X=1:WAL=500
 40 POSITION X+1,2:? #6;"[AB]";:POSITION X+1,3:?
 #6;"[CD]";:POSITION X+1,4:? #6;"[EF]";
 55 TN$(2,2)=STR$(INT(RND(1)*10))
 :BN$(2,2)=STR$(INT(RND(1)*10)):L=L+1:IF L=PR THEN GOTO 7000
 56 POSITION 2,0:? #6;W;"/";L;
 57 TN$(1,1)=STR$(INT(RND(1)*10))
 :BN$(1,1)=STR$(INT(RND(1)*10))
 58 TN=VAL(TN$):BN=VAL(BN$)
 60 POSITION X+1,20:? #6;TN$;:POSITION X,21:?
 #6;"+";BN$;:POSITION X,22:? #6;"___";
 70 POSITION X,23:? #6;"   ";:GOSUB WAL
 100 SOUND 0,8,8,4
 120 IF RND(1)>0.95 THEN FOR D=10 TO 5 STEP -1:SOUND
 1,D,10,INT(RND(1)*10):NEXT D:SOUND 1,0,0,0
 140 GOSUB 600
 180 IF RND(1)>0.95 THEN FOR D=10 TO 5 STEP -1:SOUND
 1,D,10,8:NEXT D:SOUND 1,0,0,0
 184 GOSUB 600
 185 IF RND(1)>0.95 THEN FOR D=15 TO 0 STEP -1:SOUND
 1,100,8,D:FOR E=1 TO 20:NEXT E:NEXT D:SOUND 1,0,0,0
 195 GOSUB 600
 200 GOTO 120
 500 FOR A=2 TO 17
 501 GOSUB 2000
 535 NEXT A
 540 RETURN
 600 TRAP 840:K=PEEK(764)
```

then ENTER programs when they are finished, to clear the Atari variable table of all but the variables actually being used.

Variable Table

NU$—String holding answer input by player
TN$—String holding randomly generated top number
BN$—String holding randomly generated bottom number
A$—String to hold player input to questions
P—Horizontal position of individual number input by player as an answer
START—Beginning location of character set in the operating system
NOW—Variable loop pointing to next character in the character set
CH—ATASCII number of character set
Q—Location of RAMTOP (PEEK (106))
I—Location of PLAYER/MISSILE Base Address
T—Variable for top end of volume in motion sounds
L—Number of problems completed
X—Horizontal position variable
WAL—Gorilla movement subroutine
PR—Number of problems chosen
W—Number of problems successfully completed
TN—Actual top number of problem
BN—Actual bottom number of problem
D—Decreasing pitch used in booming sound
E—Timing loop variable
A—Vertical position of gorilla or erase pattern
K—ATASCII number input from keyboard
N—Actual number input from keyboard
AMT—Answer: total of answer numbers in the ones, tens, and hundreds columns
J—Sum of top number (TN) and bottom number (BN)
C—Number of times gorilla goes through movement routine
B—Volume of motion sound routine ending in variable T, also wait routine
F—Flag set to 0: input character set. Flag set to 1: jump directly to main body of program
WT—Wait routine

The Program

Lines 0 through 4 introduce our authors, and set up our new character set. Line 2 DIMensions the various strings we will need, and makes sure that the string to hold the answer is empty.

In line 3 we find the top of our available memory by PEEKing RAMTOP, which is location (106) in memory. Then we fool the operating system into believing that

```
610 IF K=31 THEN N=1:GOTO 700
615 IF K=30 THEN N=2:GOTO 700
620 IF K=26 THEN N=3:GOTO 700
625 IF K=24 THEN N=4:GOTO 700
630 IF K=29 THEN N=5:GOTO 700
635 IF K=27 THEN N=6:GOTO 700
640 IF K=51 THEN N=7:GOTO 700
645 IF K=53 THEN N=8:GOTO 700
650 IF K=48 THEN N=9:GOTO 700
655 IF K=50 THEN N=0:GOTO 700
660 IF K=12 THEN AMT=VAL(NU$):GOTO 800
670 IF K=33 THEN NU$="     ":P=4:GOTO 705
690 RETURN
700 P=P-1:IF P<1 THEN K=33:GOTO 670
701 NU$(P,P)=STR$(N):POKE 764,255
705 POSITION X,23:? #6;NU$:RETURN
800 J=TN+BN
810 NU$="     ":P=4:POKE 764,255:POSITION X;23:? #6;NU$
820 POSITION X+1,2:? #6;"  ";:POSITION X+1,3:? #6;"
";:POSITION X+1,4:? #6;"  ";
830 POSITION X+1,8:? #6;"  ";:POSITION X,8+1:? #6;"
";:POSITION X,8+2:? #6;"   ";
835 IF AMT=J THEN WAL=500:W=W+1:GOTO 1000
840 POKE 764,255:X=8:WAL=900:GOSUB 1010:GOTO 4000
900 T=30:FOR A=2 TO 17:GOSUB 3000:NEXT A:T=20:RETURN
1000 IF X=8 THEN 5000
1005 X=1:T=10:A=17:FOR C=1 TO 20:GOSUB 2000:NEXT
C:T=20:FOR B=1 TO 200:NEXT B:GOSUB 1010:GOTO 55
1010 FOR A=17 TO 23:POSITION 0,A:? #6;"
";:NEXT A:RETURN
2000 POSITION X+1,A-1:? #6;"  ";
2005 POSITION X+1,A:? #6;"[NO]";:POSITION X+1,A+1:?
#6;"[PQ]";:POSITION X+1,A+2:? #6;"[RS]";
2006 FOR B=1 TO T:SOUND 3,200,8,B:NEXT B
2007 POSITION X+1,A:? #6;"[TU]";:POSITION X+1,A+1:?
#6;"[YV]";:POSITION X+1,A+2:? #6;"[WX]";
2008 FOR B=1 TO T:SOUND 3,200,8,B:NEXT B
2009 POSITION X+1,A:? #6;"[AB]";:POSITION X+1,A+1:?
#6;"[CD]";:POSITION X+1,A+2:? #6;"[EF]";:SOUND
3,0,0,0:RETURN
3000 POSITION X+1,A-1:? #6;"  ";
3001 POSITION X+1,A:? #6;"NO";:POSITION X+1,A+1:? #6;"PQ";
3002 FOR B=1 TO T:SOUND 3,170,8,B:NEXT B
```

the available memory is five pages smaller than it actually is, so we won't accidentally load our program on top of the changes we are going to make.

We then get an even number above our new RAMTOP (by adding 1) and multiply this number by 256. This new value of Q gives us the starting location of our new character set.

We use the number 256 because the Atari 6502 microprocessor divides memory into 256-byte "pages," and we must start the new character set at the beginning of an even page mark.

Line 4 begins the river sound and jumps to the introduction and instructions.

The subroutine at 6100 prints out the name of the program, and the authors. Then there is a pause at line 6120, so the title can be read, followed by a jump to the section asking for the number of problems desired—line 6000.

If this is the first time the program has been run, the F flag is set at 0 and a message asks the player to wait while the character set is set up. The program then returns to line 5.

The Atari character set can't actually be changed, because it is permanently embedded in ROM starting at address 57344. So we must move it into RAM. We do this in line 3 by PEEKing the character set and then POKEing it into the space we have set aside above RAMTOP.

Line 6 reads the DATA defining the new characters and POKEs it into our new locations. In line 11 we POKE the location of our new character set into location (756), just above RAMTOP in the Character Base Register.

There is a stream in our graphics jungle. In line 8 we create this by turning on two Players and setting their location just below RAMTOP then POKEing this into the Player/Missile Base address 52479.

Jumping to line 15, the horizontal position of Player 0 is POKEd into 53248, Player 1 into 53249. POKEs (704) and (705) set the color, POKEs (53261) and (53262) set up the shape and POKEs (53256) and (53257) set up Player size.

We keep track of the number of times the Gorilla finds a problem with the variable L, and use the variable X as the X coordinate of our gorilla's location. Atari Basic accepts names as well as line numbers in GOTOs and GOSUBs, so we give the movement subroutine a name, WAL.

Line 40 sets the starting location of the animal by using a position statement in X/Y coordinate form. Line 55 then randomly selects a top number which is placed in TN$ and a bottom number, placed in BN$.

```
3003 POSITION X+1,A:? #6;"TU";:POSITION X+1,A+1:? #6;"YV";
3004 FOR B=1 TO T:SOUND 3,170,8,B:NEXT B
3005 POSITION X+1,A:? #6;"AB";:POSITION X+1,A+1:?
#6;"CD";:SOUND 3,0,0,0:RETURN
4000 T=10:FOR A=17 TO 2 STEP -1:GOSUB 3000:POSITION
8,A+1:? #6;"   ";:NEXT A:T=20:GOTO 55
5000 GOSUB 1010:T=20:X=1:GOTO 55
6000 ? "}HOW MANY PROBLEMS WOULD YOU LIKE";:INPUT PR
6005 IF F=1 THEN 11
6010 ? "}JUST A MINUTE, WHILE I LET THE GORILLA
OUT OF HIS CAGE.":RETURN
6100 GRAPHICS 1:SETCOLOR 2,0,0:POSITION 6,4:?
#6;"[animath]":POKE 752,1:? "          A MATH PROGRAM BY"
6110 ? "   LLOYD OLLMANN AND JERRY WRIGHT":? "   CHARACTER
DESIGN   MIKE POTTER":POKE 752,1
6120 FOR WT=1 TO 1000:NEXT WT:? "}":GOSUB 6000:RETURN
7000 GRAPHICS 0:POKE 704,0:POKE 705,0:POKE 710,0
7010 ? "THIS TIME YOU GOT ";W;" OUT OF ";L:? "RIGHT.":?
"DO YOU WANT TO TRY AGAIN? (Y/N)":INPUT A$
7020 IF A$<>"Y" THEN END
7030 W=0:L=0:F=1:GOTO 6000
10000 DATA 0,0,0,0,1,3,6,52
10010 DATA 0,0,0,0,128,192,96,44
10020 DATA 124,254,255,239,239,231,231,199
10030 DATA 62,127,255,247,247,231,231,227
10040 DATA 71,83,115,6,6,4,2,14
10050 DATA 226,202,206,96,96,32,64,112
10060 DATA 255,255,255,255,255,255,255,255
10070 DATA 15,15,15,15,15,15,15,15
10080 DATA 240,240,240,240,240,240,240,240
10090 DATA 1,1,1,1,1,1,1,1
10100 DATA 128,128,128,128,128,128,128,128
10110 DATA 3,3,3,3,3,3,3,3
10120 DATA 192,192,192,192,192,192,192,192
10130 DATA 0,0,0,0,1,3,6,4
10140 DATA 0,0,0,0,128,198,110,63
10150 DATA 28,126,255,239,239,255,59,3
10160 DATA 63,127,255,227,227,227,231,238
10170 DATA 199,239,126,60,24,0,0,0
10180 DATA 224,224,112,48,48,60,60,0
10190 DATA 0,0,0,0,1,99,118,252
10200 DATA 0,0,0,0,128,192,96,32
```

Line 56 places the number of problems successfully answered next to the number of problems tried, and line 57 gets more numbers for the number strings. Line 58 gets the value of the strings and places them into variables TN and BN.

Line 60 places the numbers in their proper positions at the bottom of the screen.

In line 60 we jump to the gorilla animation section. This time WAL=500 so in line 500 we find the vertical positions for the gorilla in a FOR/NEXT loop and jump to the actual movement subroutine at line 2000.

Lines 2000 through 2009 draw the gorilla and move his arms and legs, while making the movement sound. Then the subroutine jumps back to line 501 where it gets a new position from the variable A in the FOR/NEXT loop. It then goes back to the movement routine until it reaches vertical position 17 on the screen. Next we jump back to line 100 for a sound routine and then jump to line 600.

This is the keyboard routine—where we PEEK location (764) to find the internal code of the last key pressed. The computer runs through a series of IF/THEN statements to determine which key has been pressed by the player, and compares it to a list of valid inputs. The first number input goes into the ones column, the second into the tens column, the third into the hundreds column, and the fourth into the thousands column.

This is done by setting up a number holding string (NU$). The position of the number in the string is determined by line 700, which starts with P=4, so the first number is placed in the fourth position of NU$, the second number in the third position and so on. Line 705 prints the NU$ on the screen and then the Atari loops around to line 120 and back through the keyboard routine until Return is pressed at line 660.

Line 670 allows you to recover if a mistake is made. All you have to do is press Delete/Backspace or the Space Bar, the NU$ is cleared, and you start back at the ones column. You then repeat the procedure until you have what you feel is the correct answer.

Pressing Return takes you out of the loop at line 660 where this time the value of the numbers in NU$ is transferred to the variable AMT. We then jump to 800 to find out if the answer is correct.

Line 810 clears NU$ and resets location (764) by POKEing in 255.

Lines 820 and 830 blank out the standing gorilla, then 835 determines if the answer is right. If it is, the gorilla walking routine (WAL) jumps back to line 500, adds 1 to the amount answered correctly, and jumps to line 1000.

If AMT doesn't equal J, then the gorilla movement subroutine is set to 900, the horizontal position (X) is moved over 8 places, and the gorilla runs through the subroutine at line 400 which forces him into the river. A reverse FOR/NEXT loop carries him downstream in subroutine 3000. Then the program jumps back to line 55 where the new value of X swims him upstream with the subroutine at line 900, and gives him a new addition problem to answer.

If the question was answered correctly, the program jumps to the subroutine at 2000 through 2009, where the gorilla jumps up and down with joy. The routine then sets the sound volume variable T to 20 and the horizontal position variable X to 1 and then jumps back to the main program loop at line 55.

When the number of problems chosen (PR) equals the number actually done (L), the program goes to line 7000 where it displays a score and offers a chance to play again.

We hope this program and accompanying explanation have given you an idea of some of the things that can be achieved with the Atari Personal Computer. We enjoy this system thoroughly, and hope that many more people will soon see the Atari as a computer with truly incredible possibilities.

Instructions

After the program is loaded and the player has chosen the number of problems, the gorilla will come on the screen and run down to the first problem. The answer should be typed in with the first number in the ones column, the second in the tens column, and the third in the hundreds.

If you make a mistake, just press the space bar. When you have the correct answer, press the return key. The gorilla will tell you if you are right.

In the listing that follows, several characters are in square brackets. These should be typed in as inverse characters. The Epson MX-80 prints a "] " instead of a clear sign, so when you see that symbol type Escape, then Control and Clear together. □

Greater Graphics Control

Marni Tapscott

The Atari has nine graphics modes. Modes 1 through 8 have a split screen, however, the split screen may be overridden by adding 16 to the mode number. Modes 1 and 2 are text modes with five colors. Characters in graphics mode 1 are twice as high and twice as wide as those in mode 0. Characters in mode 2 are twice as high and twice as wide as those in mode 0.

If you have ever tried to use the Atari graphics characters in mode 1 or 2 only to be dismayed by a screen full of hearts, or have had difficulty using all five colors available in those modes, read on. Solutions to some of the problems encountered in both areas will be discussed.

The character set in graphics mode 0 has 128 characters, upper and lower case letters, punctuation, numbers and Atari graphics characters. However, in graphics modes 1 and 2, only 64 characters are available at a time. There are three choices: numbers, upper case letters and punctuation including a blank space; the Atari graphics characters and lower case letters with no blank space; or your own character set.

Creating Blank Spaces

Frequently, you will want to use blank spaces as well as the graphics characters. There are two ways of creating blank spaces. One is to give up one of the five colors available; simply make color register 0 the same color as the background and proceed to plot other characters using only color registers 1, 2 and 3. This is the straightforward solution. The short program in Listing 1 illustrates this alternative.

The second method of creating blank spaces requires more work; one character must be redefined. Novice programmers may be put off by the imposing sound of "redefining a character set," but I have discovered that it is not difficult and that it can open the door to greater graphics control and creativity.

It is important to point out that one or several characters can be redefined without redefining the whole character set. There are four basic steps.

First, we must allocate space in RAM for the character set and protect it from Basic. The top of RAM is the end of the section of memory accessible to the user. The physical top of RAM is stored in a location called RAMTOP. The area above the value stored in RAMTOP is Read Only Memory or ROM which contains permanent storage of programs and data that may never be changed. The operating system, for example, is stored here.

If we store a lower value in RAMTOP, we effectively reserve a section of RAM. The operating system will be fooled into thinking less RAM memory is available, and we can keep our new character set from being changed or erased by storing it in this area.

When I refer to "up" in memory, I am referring to those memory locations with higher numbers; "down" refers to memory locations with lower numbers. The diagram in Figure 1 may help.

Step one: Reserve memory for the new character set. Graphics modes 1 and 2 require 512 bytes or two pages for redefining a character set. In mode 0, we need 1024 bytes or four pages to redefine the 128 characters available. We PEEK at what is stored in RAMTOP (location 106), subtract the appropriate number of pages (each page=256 bytes) from that value and POKE it back into 106.

Step two: Move the present character set from ROM into the reserved section of memory. This is easily accomplished with a FOR/NEXT loop PEEKing the character set in the ROM location and POKEing it into the new location. The character set containing upper case letters, numbers and punctuation is located at 57344 in ROM and the alternate set containing the graphics characters is located at 57856 in ROM.

Step three: Inform the operating system where the new character set is located with a POKE 756, X where X equals the address of the new character set. Every time a graphics statement or reset is executed, the value in location 756 is reset to 224, the starting page address of the old character set in ROM, so it is best to include this POKE statement after any graphics mode statement.

Step four: Redefining the characters. The definition of a character uses 8 bytes in memory. Eight 0's must be poked into memory to take the place of an existing character. Since the heart is the first character in this set, I found it easiest to replace. The first 8 bytes or locations 0 through 7 in the section of memory we have set aside contains the heart. If we POKE 0's into these locations we will finally have a blank space. Incidentally, the reason the screen fills with hearts in

Marni Taspscott, 297 Missouri St., San Francisco, CA 94107.

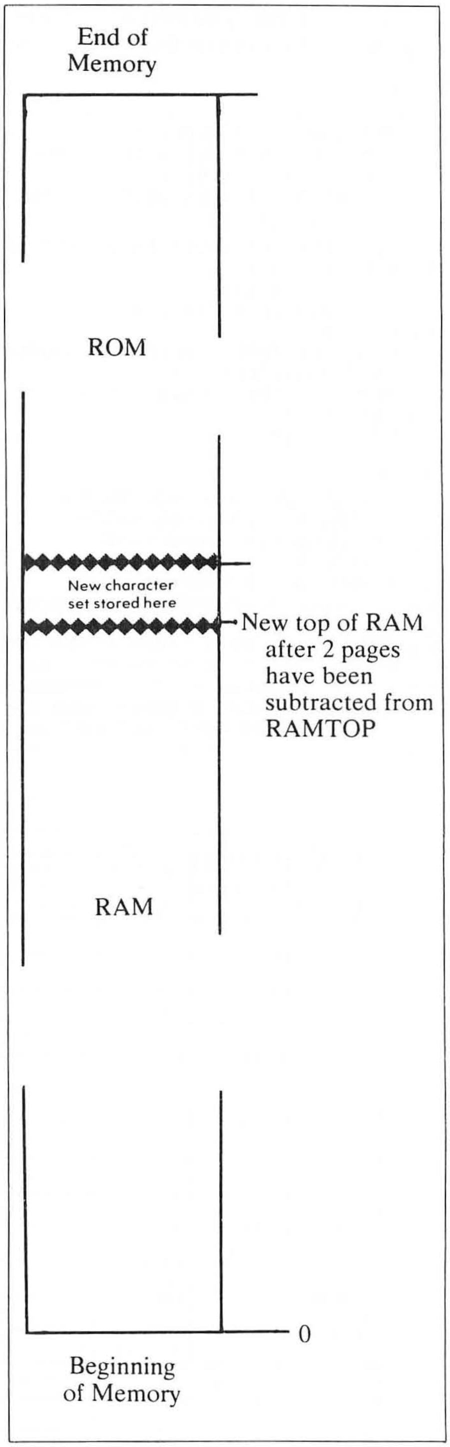

End of Memory
ROM
New character set stored here
New top of RAM after 2 pages have been subtracted from RAMTOP
RAM
0
Beginning of Memory

Figure 1.

Listing 1.

```
110 GRAPHICS 1:POKE 756,226
120 ? "THIS IS WHAT HAPPENS WHEN 756 IS POKED WITH 226 IN GR. 1"
130 FOR WAIT=1 TO 2000:NEXT WAIT
140 SETCOLOR 0,0,0:REM SET COLOR REGISTER 0 TO SAME COLOR AS BACKGROUND
150 ? "THIS IS WHAT HAPPENS WHEN A COLOR REGISTER IS MADE SAME COLOR AS
    BACKGROUND."
160 FOR WAIT=1 TO 2000:NEXT WAIT
```

modes 1 and 2 when you are trying to use the graphics characters is that the heart is stored in the same relative position as the blank space in the other character set.

Listing 2.

```
100 REM CHARACTER REDEFINITION
110 REM STEP ONE: SET ASIDE MEMORY FOR CHARACTER SET
120 POKE 106,PEEK(106)-2
130 GRAPHICS 2+16:REM GR.STMT.HERE PREVENTS OVERLAP OF DISPLAY LIST
    & CHARACTER SET
140 REM STEP TWO MOVE: CHARACTER SET INTO NEW LOCATION
150 A=PEEK(106)*256
160 FOR B=0 TO 511
170 POKE A+B,PEEK(57856+B)
180 NEXT B
190 REM STEP THREE: POKE NEW ADDRESS OF CHARACTER SET
200 POKE 756,PEEK(106)
210 REM STEP FOUR: CHANGE HEART TO BLANK SPACE
220 FOR C=0 TO 7
230 POKE A+C,0
240 NEXT C
250 REM
310 REM SET UP COLOR REGISTERS
330 SETCOLOR 0,13,8:REM GREEN
340 SETCOLOR 1,4,8:REM PINK
350 SETCOLOR 2,10,8:REM TURQUOISE
360 SETCOLOR 3,2,8:REM GOLD
365 SETCOLOR 4,12,4:REM BACKGROUND COLOR TO GREEN
370 REM
390 COLOR 60:PLOT 5,5:REM GREEN ARROW
400 COLOR 28:PLOT 6,5:REM PINK ARROW
410 COLOR 188:PLOT 7,5:REM TURQUOISE ARROW
420 COLOR 156:PLOT 8,5:REM GOLD ARROW
450 GOTO 450:REM KEEPS DISPLAY ON SCREEN
```

Figure 2.

Column 1				Column 2				Column 3				Column 4			
#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR
0	Space	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	[1] ␛	107	k	123	♠
12	,	28	<	44	L	60	\	76	▘	92	↑	108	l	124	|
13	—	29	=	45	M	61	]	77	▔	93	↓	109	m	125	[1] ↖
14	–	30	>	46	N	62	∧	78	▁	94	←	110	n	126	[1] ◀
15	/	31	?	47	O	63	—	79	▖	95	→	111	o	127	[1] ▶

Figure 3.

Table 9.7—CHARACTER/COLOR ASSIGNMENT

		Conversion 1	Conversion 2	Conversion 3	Conversion 4
MODE 0	[2]SETCOLOR 2	# + 32	# + 32	# - 32	NONE
		POKE 756,224		POKE 756,226	
MODE 1 OR MODE 2	SETCOLOR 0	-#32	# + 32	# - 32	# - 32
	SETCOLOR 1	NONE	# + 64	# - 64	NONE
	SETCOLOR 2	# + 160	# + 160	# + 96	# + 96
	SETCOLOR 3	# + 128	# + 192	# + 64	# + 128

Charts provided courtesy Atari Inc. © 1980.

These four steps eliminate the heart and define a blank space. Now we are ready to assign colors and positions to characters.

Assigning Color and Position

There are two methods; we may use either the POSITION and PRINT #6 statements or the COLOR and PLOT statements. Color manipulation is less obvious when using POSITION and PRINT #6 statements.

The ATASCII number that corresponds to both the character and the color desired must be obtained through some experimentation.

Since the other method employs charts already available in the Atari Basic Reference Manual, this method will be described in greater detail. For convenience the charts from pages 55 and 56 in the Atari Basic Manual have been reproduced here.

First, the four colors desired are established in the color registers using SETCOLOR statements. SETCOLOR 0,1,8 establishes gold in register 0. Next, find the character you wish to use in the chart in Figure 2. Make note of both the number next to the character and the column in which it is located. Looking at the second chart in Figure 3, add or subtract the number listed here according to the color desired. The "columns" on the first chart correspond to the "conversions" on the second chart.

For example, I want a gold up arrow to appear at Row 5, Column 5. The up arrow is 92 in Column 3 in Figure 2. Looking at Figure 3, we subtract 32 from 92 since gold is in color register 0. The statement below accomplishes our goal:

COLOR 60:PLOT 5,5

Listing 2 is a short program which illustrates both the redefinition of the heart character to a zero and the use of SETCOLOR, COLOR and PLOT statements for full use of all five colors. (The fifth color is the background color.) One word of caution regarding running the program: always press the system reset button before re-running because the system continues to subtract pages in memory until it interferes with the display memory.

Suggestions for further experimentation are:

- Redefine more characters for greater graphics variety.
- Combine two or four or more characters for a larger, more complex shape.
- Animate shapes through color rotation.
- Animate shapes through redefinition of a figure (animal, person) in several positions and rotation of positions.

The Atari Graphics Composer

David Lubar

Everyone who has come within thirty feet of an Atari knows that the machine is capable of great graphics. Everyone who has come closer than that knows how tough it is to get those great graphics. By producing the *Atari Graphics Composer*, Versa Computing has taken care of the hard work, leaving the user free for creativity and experimentation. This set of utilities performs five main functions; hi-res drawing, medium-res drawing, text writing, geometric figure creation, and player creation. The combination is powerful enough to allow a wide range of graphics.

The high-res mode allows drawing with paddles or joystick on a four-color screen with a resolution of 320 by 160. There is one background color, which can be changed at any time, and three foreground colors. While the luminance of the foreground colors can be changed, the color value is predetermined by the background. In this mode, the user can either draw freestyle, or draw lines between any two points. Other options include fill and brush routines. There are two types of brushes; normal brushes fill an area with a solid pattern, the air brush puts a pattern of dots over an area. Combining these, one can color in a picture, then add shading. The fill routine, written in Basic, is not fast, but it is very thorough, filling in most irregular patterns without missing any spots.

SOFTWARE PROFILE

Name: Atari Graphics Composer

Type: Graphics utility

System: Atari 400 or 800, 32K RAM, Basic Cartridge, paddles or joystick.

Format: Disk or Tape

Language: Basic and Machine Language

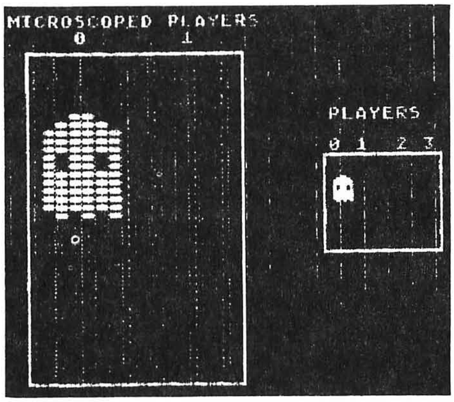
Summary: Versatile system for graphic creation

Price: $39.95 on disk or tape

Manufacturer:
Versa Computing, Inc.
3541 Old Conejo Rd. Suite 104
Newbury Park, CA 91320

Another nice feature is the accelerating crosshair. When the joystick is moved to a new position, the crosshair moves slowly at first, then speeds up. This allows for fine control over a small area and less waiting time when crossing the screen. While the quality of any graphics done in this mode depends, obviously, on the user's artistic ability, the capability is there to produce detailed pictures.

The medium-res mode provides a screen with 160 by 80 resolution, with one background and three foreground colors. These colors can be changed at any time. (For those unfamiliar with the Atari, a change in color actually changes a color register, thus not only do future lines appear in that new color, but lines drawn previously with that color also change to the new color.) As with the hi-res mode, medium-res also provides a fill routine and a selection of brushes.

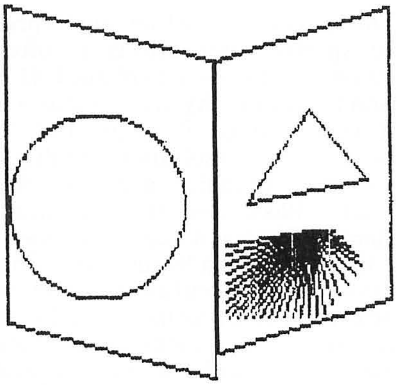


Player creation is now a simple and dynamic process.

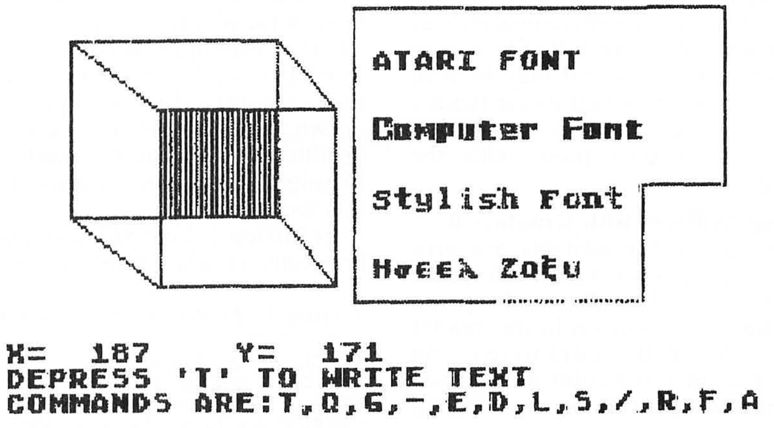
The text mode places characters from any of four fonts on the hi-res screen. In the disk version of this package, users can switch between any of the modes using hi-res without losing the picture on the screen. Thus a scene can be drawn using the drawing mode, then labeled in the text mode. Along with upper and lower case, all special Atari symbols are supported. Also, the program will accept any user-generated fonts, though the documentation doesn't cover the process of font creation.

To write on the screen, the user first positions the cursor at the desired starting point, using joystick or paddles, then types "T" for text. From that point until the escape key is pressed, all typed characters will be displayed on the screen. Editing keys such as delete still perform their usual function. If the user has switched to lower case, the program won't recognize any commands, but it will prompt the user to press the SHIFT and ALL CAPS keys.

The geo-maker mode allows the creation of a variety of geometric figures, from circles and arcs to triangles and parallelograms. Figures are defined by specifying points. A circle, for example, is defined by its center and any edge point. Triangles and parallelograms require three points. The circle and arc take the longest creation time, while other figures appear rapidly. The geo-maker includes a routine for Moire patterns. The user specifies the step value and, if desired, a window area, then uses



Figures and Moirê pattern made with the geo-maker.



ATARI FONT

Computer Font

Stylish Font

Hαεεk Zoξu

```
X=  187     Y=  171
DEPRESS 'T' TO WRITE TEXT
COMMANDS ARE:T,Q,G,-,E,D,L,S,/,R,F,A
```

Cube was done using the draw-to and fill routines of the hi-res mode. Lettering was added in the text mode.

David Lubar is a former associate editor for *Creative Computing* **magazine.**

the joystick or paddles to fill an area with the pattern.

One of the most attractive features of the Atari is the ability to use players in animation. These shapes are usually coded by hand. The *Graphics Composer* has automated the process. Player creation is potentially the most valuable utility on the disk. It presents the user with a grid for designing players. Each large dot turned on in the grid is also displayed in true size on the screen. Once a player is created, it can be saved, and the decimal values representing the player can be displayed, allowing the user to put that player in his own programs.

Beyond explaining all the functions of the programs, the documentation also describes how to use the picture loading routine in other programs, thus making pictures created on this system retrievable by other software.

Anyone doing, or planning to do, graphics work on the Atari should seriously consider the *Atari Graphics Composer*. □

Artifacting With Graphics 7-Plus

Harry G. Arnold

The technique of artifacting to produce special color effects is often mentioned but seldom explained. One reason for the brevity in instructions is that even though it is possible to achieve high resolution, multiple color, and graphics displays with artifacts, Basic and the Operating System (OS) do not support artifacting in a user-friendly manner. This discussion will present an introduction to television color artifacts similar to that seen in many other places, then proceed to describe a method and program listings to more easily use artifacts in Atari Basic. Because the resolution of the resultant display is half way between GR. 7 and GR. 8 it is often referred to (affectionately) as "Graphics 7-Plus" or GR. 7+. It is in fact Antic mode 14 and may be seen in its more refined and domesticated incarnations in several high-resolution games (particularly those with the weird colors.)

Making Artifacts with Graphics 8

Television color artifacts are produced when a color cell on the screen containing a red, a blue, and a green dot is hit by an electron beam smaller than the cell. If the beam were as big as the cell all three color dots would glow producing a spot on the screen in one of Atari's 128 (or is it now 256?) colors. When for some reason the beam only hits half of a color cell, one of two colors shows up, depending on which half of the cell was hit. These two colors will usually be some shade of yellow/green and some shade of blue/purple (with red/brown possible) depending on which background color was specified. One reason that only half of a color cell may be hit is that the horizontal resolution of GR. 8 is equal to half of a color cell. You probably have seen this effect show up as multi-colored lines when drawing in GR. 8. If you were to observe the GR. 8 colors closely you would find that all of the odd-numbered horizontal pixels are one color while all the even-numbered ones are a second color. These two colors are the artifact colors. Two lines plotted and drawn side-by-side (vertically) will produce a third color, usually white. Listing 1 demonstrates artifacting by this technique.

In Listing 1, Line 520 draws vertical lines only on odd-numbered pixels and Line 530 only on even-numbered ones, while Lines 540 and 550 draw two lines together to produce the three artifact colors (on a fourth background color.) To experiment with the variations in color, change the SE. 2,0,0 in Line 500 to different values. Table 1 lists some approximate colors that are possible. If your computer was built before 1982 and does not have the GTIA upgrade the colors will be different (probably reversed).

Listing 1. Artifact Colors using GR. 8.

```
500 GR. 8: SE. 2,0,0: COLOR 1
510 FOR I= 0 TO 20 STEP 2
520 PLOT 41+ I, 50: DR. 41+I,100
530 PLOT 80+ I, 50: DR. 80+I,100
540 PLOT 120 + I, 50: DR. 120+I,100
550 PLOT 121 + I, 50: DR. 121+I,100
560 NEXT I
```

So, there it is. Four colors in GR. 8, right? Well, not quite. Since we only plotted every other pixel or used two together to produce the colors each vertical line is twice as wide as the normal GR. 8 line. Thus the horizontal resolution was cut in half to 160 pixels — the same as GR. 7. The vertical resolution did not change, however, and remained at 192 pixels — the same as GR. 8. Hence the nickname GR. 7+. The colors are also in between those of GR. 7 and GR. 8, for even though there are four colors as in GR. 7, they are all controlled by one register as in GR. 8. Still it is a multiple color, high resolution graphics mode and we just accessed it with Basic. It is a different graphics mode than any we could normally access.

Drawing and Filling in Graphics 7-Plus

The above method is somewhat cumbersome when many different

Harry G. Arnold, 109 Newhaven Road, Oak Ridge, TN 37830.

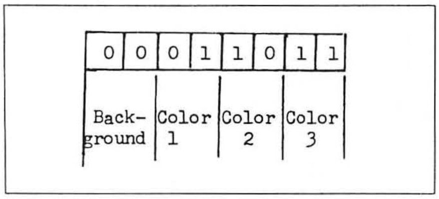
shapes and even simple diagonal or horizontal lines are to be drawn. So, let's experiment with ways to use the OS DRAW and FILL routines in GR. 7-Plus even though they were not made for it.

Listing 2. Drawing 3 Colors with GR. 7 for GR. 8 Display.

```
500 GR. 8:GR. 7:REM GR. 8 Just clears out old garbage first
510 COLOR 1:PLOT 50,60:DR. 50,30
520 DR. 25,30:COLR=1:POS. 25,60:GOSUB 580
530 COLOR 2:COLR=2:PLOT 100,60
540 DR. 100,30:DR. 70,30
550 POS. 70,60:GOSUB 580
560 COLOR 3:COLR=3:PLOT 150,60
570 DR. 150,30:DR. 125,30:POS. 125,60:GOSUB 580:GOTO 590
580 POKE 765,COLR:XIO 18,#6,0,0,"S:":RETURN
590 END
```

One way to use the DRAW and FILL routines is to use GR. 7 to create the screen display, then switch to GR. 8 to display it. When GR. 7 creates data for a colored pixel on the screen it uses two bits of display memory for each pixel. In Figure 1 the options available with four pairs of these bits are shown (0 means "off", 1 means "on"). Two bits together that are both "off" produce the background color, while any one or both bits of a pair that are "on" will produce color from one of the three color registers in GR. 7. When the GR. 7 display list fetches 40 bytes for display, with 8 bits per byte, it gets 320 bits. But since each pixel in GR. 7 requires two bits, GR. 7 only produces 160 pixels per line from the 320 bits per line in memory — but each pixel can be one of four different colors.

Figure 1. Bit pairs in GR. 7 to produce colored pixels.



When GR. 8 creates the display data it only stores one bit per pixel. When the GR. 8 display list gets 40 bytes of data it will produce the full 320 pixels per line with that data. Thus, with only one bit per pixel to work with, no color information can be included. A pixel is either on or off because a bit is either on or off. Thus, GR. 8 would interpret the data represented by Figure 1 one bit at a time and simply display a light or a dark pixel according to the status of each bit. But notice in Figure 1 that each pair of GR. 7 bits corresponds to either an even- or an odd-numbered pixel in GR. 8. Therefore GR. 8 would interpret the GR. 7 display data as artifact colors, just as it did the lines drawn on odd or even pixels in Listing 1. Each GR. 7 Color command corresponds to one of the GR. 8 artifact colors.

Let's draw a display with GR. 7 then change to GR. 8 without erasing the display. To do this we will simply add 32 to the Graphics mode number (e.g. GR. 8 + 32). Listing 2 will draw and fill three colors in GR. 7 (then we will add a line to change to GR. 8). Now, after examining the display, add the following line and run again.

```
590 GR. 8+32:SE. 2,0,0
```

Nobody is going to claim that this is of much use as it looks, but we have in fact used the DRAW and FILL routines to create four colors in GR. 7 that were interpreted as four artifact colors in GR. 8. If we could just clean up the trash and center the display we could draw the bottom half of a GR. 8 screen using the GR. 7 display memory and Basic Graphics statements.

Now, let's do it the other way around. The display data are sent to the screen by a special microprocessor called Antic. It knows where to find data, how much to send, and in which graphics mode because of the display list whose address is stored in memory at locations 560 and 561. The OS on the other hand puts the display data in memory. It knows where to put it because the screen memory address is stored in locations 88 and 89. It knows which graphics mode because the mode number is stored in memory at location 87. If we use the GR. 8 command to set up a Graphics 8 display list, Antic will fetch each line of memory and display it as GR. 8 data—each pixel either on or off, no color. If we POKE 87,7 the OS will put the display data into memory as GR. 7 data with the appropriate color bits set. This sounds exactly like the situation we encountered when explaining Figure 1, doesn't it? Add Listing 3 to Listing 2 to see this effect.

The end result is the same as before with one important exception; the display started at the upper left hand corner of the screen this time, and was therefore more controllable. Since the GR. 8 display list only generates one line of TV scan per pixel and GR. 7 data were created assuming there would be a GR. 7 display list to generate 2 lines per pixel, there were only enough data to fill the upper half of the screen. However, the bottom half was still filled with the data we generated before. Even though the bottom half still needs fixing, let's just observe this important point for the time being: it is possible to fill a Graphics 8 screen (i.e. use a GR. 8 display list) with Graphics 7 display data.

Listing 3.
Adding GR. 7 Data to Listing 2.

```
590 IF J>0 THEN END
600 GR. 8+32
610 POKE 87,7
620 SE. 2,0,0:COLOR 1
630 J=1:GOTO 510
```

A More Useful DRAW for Graphics 7-Plus

Let's examine what happens when we run Listings 2 and 3. Graphics 8 requires 7680 bytes of display data. Graphics 7 only requires half that amount or 3840 bytes because its display list is supposed to tell Antic to generate twice as many display lines on the screen as Graphics 8 does. By setting up a Graphics 8 display using the GR. 8 command, 7680 bytes of memory are reserved for the display data. However, when we draw the display using the Poke to location 87, we are generating Graphics 7 display data in memory, so only 3840 bytes are stored in memory. The result is that the display only fills the upper half of the screen. How can we fill the remaining 3840 bytes? We did it earlier using the Graphics 7 to draw then switching to Graphics 8 without erasing the memory. Let's find a better way.

When we first set up a Graphics 8 display the OS reserves 7680 bytes of display memory. It also puts the location of the upper left screen corner data in memory locations 88 and 89 and in the display list. The OS then proceeds to consult locations 88 and 89 whenever it stores display data in memory, while Antic consults the display list when it retrieves data to display on the screen. Under standard conditions these two sets of screen data addresses are the same. However, if we were to set up a Graphics 8 display, then change memory locations 88 and 89 to show the address of the upper left corner of the screen to be 3840 bytes lower than the Graphics 8 screen corner, it should be possible to draw on the bottom half of the Graphics 8 screen since Antic still finds the corner of a display twice as big. Since the Graphics 8 display is twice as big as the Graphics 7 as far as memory is concerned, the effect of moving the display pointer for the Draw and Fill commands is to cause draw and fill to occur on the bottom half of the screen. To draw on the top half of the screen again, we simply Poke the original values back into locations 88 and 89.

Now, let's see if we can put it all together and draw a complete Graphics 8 screen with Graphics 7 data. We will set up a Graphics 8 display list with the GR. 8 command which will reserve 7680 bytes for display memory. Then we will tell the OS to put Graphics 7 data into that memory with a POKE 87,7 just as in listing 3. We will then proceed to draw the upper half of the screen display using Graphics 7 BASIC statements, then we will add 3840 bytes to the location of the upper left corner of the screen data and Poke this into locations 88 and 89. After that, the OS will think that the middle of the screen is the upper corner and will do any further drawing on the bottom half. Since the Graphics 8 display list was unchanged, Antic fetches display data from the original left corner all the time even though we change the locations for putting data into memory. Listing 4 is an example of this procedure. (Type NEW just to ensure that previous listings are erased.)

Listing 4. Drawing on Both Halves of a Graphics 7-Plus Screen.

```
  10 DIM D88(1),D89(1)
  20 ATOP=0:BOTTOM=1:GOTO 500
 500 GR. 8+16:POKE 87,7
 510 D88(ATOP)=PEEK(88):D89(ATOP)=PEEK(89)
 520 BOTCORNER=D88(ATOP)+256*D89(ATOP)+
       3840
 530 D89(BOTTOM) = INT(BOTCORNER/256)
 540 D88(BOTTOM) = BOTCORNER -
       256*D89(BOTTOM)
 550 DRAW = 300:APLOT = 300:FILL = 400
 560 SE. 2,0,0
 570
 580
 590
```

Now continue the listing with these lines to draw on the screen.

```
 600 COLOR 1:PLOT 0,0:DR. 79,95
 610 COLOR 2:PLOT 159,0:DR. 79,95
 620 COLOR 3:PLOT 79,0:DR. 79,95
 630 POKE 88,D88(BOTTOM):POKE 89,
       D89(BOTTOM)
 640 COLOR 1:PLOT 79,0:DR. 159,95
 650 COLOR 2:PLOT 0,95:DR. 79,0
 660 COLOR 3:PLOT 79,0:DR. 79,95
9999 GOTO 9999
```

Now, the only problem we have left is where to make the line segments coincide. (Notice that all PLOT, DRAW, and POSITION commands will be limited to the Graphics 7 cursor range of 0-159 horizontal by 0-95 vertical.) However, curved lines and diagonal lines that are off-center sound like the kind of dirty drudgery work a computer was made for. When it comes to matching the upper and lower half of the screen, why don't we just let the computer do our bookkeeping? Listing 5 (lines 200-350) will accomplish this bookkeeping (it also contains a repeat of Listing 4 for reference: lines 500-560). Do not run it yet, because some minor adjustments obviously must be made to draw and plot with it. To make these adjustments, add the following lines:

```
600 DEG:COLR=1
610 FOR I=0 TO 360 STEP 5
620 X=I/3:Y=80*SIN(I)+95
630 GOSUB DRAW
640 NEXT I
650 REM Just in case the old listing
      wasn't erased first
660
```

Now it is possible to draw lines all over the screen in a transparent manner with only minor adjustments to the normal PLOT and DRAW procedures. Instead of typing:

```
COLOR 1:X= 1:Y= 2:DRAWTO X,Y
```

We type (as a numbered line):

```
COLR= 1:X= 1:Y= 2:GOSUB DRAW
```

Notice that it is not necessary to use APLOT on the very first point to be plotted. However, any time we wish to skip to a new location to continue drawing we simply type (again only as a numbered line):

```
X= 1:Y= 1:GOSUB APLOT
```

(APLOT and COLR are used because COLOR and PLOT are "reserved" words in Basic and will result in error messages.)

Let's try our original crossed lines with this new listing. Replace lines 600-640 as follows:

```
 600 COLR=1:X= 0:Y= 0:GOSUB DRAW
 610 X=159:Y=190:GOSUB DRAW
 620 X=159:Y=0:COLR=2:GOSUB APLOT
 630 X=0:Y=190:GOSUB DRAW
 640 COLR=3:X=79:Y= 0:GOSUB APLOT
 650 X=79:Y=190:GOSUB DRAW
9999 GOTO 9999
```

A detailed explanation of Lines 200-350 is a bit involved. Suffice it to say that any time a DRAW command crosses to a different half of the screen (upper or lower) the crossover point must be computed and the line must be drawn to the crossover point in the current screen half. Then the other half of the screen must be Poked into locations 88 and 89 after which the crossover point is PLOTted on the new half and the DRAW proceeds to its original destination. At no time is the value of Y for the PLOT and DRAW commands allowed to exceed 95.5, even though the computer operator is allowed to let Y range up to 191.

Just to check everything out, try adding these lines to the previous listing.

```
600 GOTO 1000
1000 DEG:J= 1
1010 COLR= J
1020 FOR K= 0 TO 315 STEP 45
1030 FOR I= 0 TO 950 STEP 10
1040 Y= I*SIN(I-K)/13+ 95
1050 X= I*COS(I-K)/23+ 75
1060 IF I= 0 THEN GOSUB APLOT:NEXT I
1070 GOSUB DRAW
1080 NEXT I
1090 J= J+ 1:IF J> 3 THEN J= 1
1100 COLR= J
1110 NEXT K
9999 GOTO 9999
```

Brief explanation of Listing 4:

```
Line 500 Sets up the Graphics 7-Plus display
Line 510 Finds the upper left screen corner used by the OS
Line 520 Converts the two bytes into a decimal value then
         adds 3840 to find the lower half's upper corner
Lines 530-540 Convert the decimal value for the lower half
         into a low byte and a high byte
Lines 570-590 Just in case you did not type NEW
Lines 600-620 Draw on the upper half (Cursor range--159X95)
Line  630 Pokes the lower half into OS memory
Lines 640 to 660 draw the lower half (Cursor range--159X95)
Line 9999 is necessary because we used the whole screen option
          in line 500
```

FILLING in Graphics 7-Plus

"So, what good are lines?" you ask, "I want color-filled areas." As you might expect the previously explained techniques can be extended to provide a semi-transparent FILL command. Notice the weasel-word qualifier on "transparent." As you are probably aware, FILL (i.e. XIO 18,#6,0,0,"S:") only works according to rules established on some level of thought other than our own. When we try to make it behave predictably under ordinary circumstances we can expect some difficulty, let alone when trying to draw on two different screens (or halves of screens) at once. So, with that caveat let us proceed.

Listing 6 shows lines to add to listing 5 in order to make FILL cross screen halves in Graphics 7-Plus. Again a detailed explanation would expend a lot of words just to say that we must compute the crossover points in between the two halves of the screen, fill to them, Poke the other half of the screen into memory at locations 88 and 89, then fill on the remaining half. Most of the tests are simply to keep the cursor within range under a variety of combinations of possibilities.

In this case, we gained some convenience, however, which will help make up for the FILL glitches that crop up from time to time. POSITION and the POKE to 765 are now automatic and XIO has been replaced.

To use the FILL command you must first draw the right and upper sides using the GOSUB DRAW and GOSUB APLOT method as before. Then, instead of typing

```
POS. 0,159:POKE 765,1:XIO 18,#6,0,0,"S:"
```

we simply type (as a numbered line)

```
X=0:Y=159:GOSUB FILL
```

The color will be the last value assigned to COLR and the X and Y values must be the lower left corner of the area to be filled. Add these lines to the previous listing to observe the fill in operation.

```
600 SE. 2,0,0:COLR=1
610 X=159:Y=190:GOSUB APLOT
620 X=159:Y=0:GOSUB DRAW
630 X=0:Y=0::GOSUB DRAW
640 X=159:Y=190:GOSUB FILL
650 X=0:Y=190:COLR=2:GOSUB APLOT
660 X=78:Y=95:GOSUB DRAW
670 X=0:Y=1:GOSUB DRAW:X=0:Y=190:
    GOSUB FILL
680 COLR=3:X=158:Y=190:GOSUB APLOT
690 X=79:Y=96:GOSUB DRAW
700 X=0:Y=190:GOSUB FILL
710 GOTO 710
```

Listing 5: Plot and Draw Subroutines for Graphics 7-Plus.

```
10 DIM D88(1),D89(1),QX(1),XFLAG(1):XF
LAG(0)=0:XFLAG(1)=0
198 GOTO 500
199 REM ****DRAW SUBROUTINE****
200 COLOR COLR:QX3=X:QY3=Y:Y=INT(QY3+0
.5)
205 IF Y>95.5 THEN QN=1:IF QY1<95.5 TH
EN QN=0:GOTO 225
210 IF Y<95.5 THEN QN=0:IF QY1>95.5 TH
EN QN=1:GOTO 225
215 DRAWTO X,Y-96*QN:QX1=X:QY1=Y
220 RETURN
225 QX2=X:QY2=Y
230 GOSUB 265:X=INT(X+0.5):GOSUB 280
235 DRAWTO X,Y-96*QN:QN=1-QN
240 POKE 88,D88(QN):POKE 89,D89(QN)
245 GOSUB 265:X=INT(X+0.5)
250 PLOT X,95-95*QN
255 DRAWTO QX2,QY2-96*QN
260 QX1=QX2:QY1=QY2:X=QX3:Y=QY3:RETURN
265 Y=95+QN:X=((Y-QY1)*(QX2-QX1)/(QY2-
QY1))+QX1
270 IF ABS(QY2-QY1)=1 THEN X=(QX2+QX1)
/2
275 RETURN
280 IF XFLAG(QM)=0 THEN XFLAG(QM)=1:QX
(QM)=X:QM=1-QM:XFLAG(QM)=0
285 RETURN
299 REM ****APLOT SUBROUTINE****
300 COLOR COLR:QX3=X:QY3=Y:Y=INT(QY3+0
.5)
310 IF Y>95.5 THEN QN=1:GOTO 330
320 QN=0
330 POKE 88,D88(QN):POKE 89,D89(QN)
340 PLOT X,Y-QN*96:QX1=X:QY1=Y:DRAW=20
0:X=QX3:Y=QY3
350 RETURN
499 REM ****SET UP GR.7+ SUBROUTINE***
500 GRAPHICS 8+16:POKE 87,7
510 D88(0)=PEEK(88):D89(0)=PEEK(89)
520 DISP2=D88(0)+256*D89(0)+3840
530 D89(1)=INT(DISP2/256)
540 D88(1)=DISP2-256*D89(1)
550 DRAW=300:APLOT=300:FILL=400
560 SETCOLOR 2,0,0:COLOR 1:COLR=1
```

The background color can be produced in the same manner as the other three colors. It is important that the only COLOR specifications be made with the COLR= statement from this point on in using the listings.

To experiment with how the different colors look when plotted on one another, remove Line 710 and (if you left lines 1000-9999 in from the previous listings) observe how the spiral behaves as it crosses the different fill areas. Also, change Line 1010 (1010 COLR=0).

Five Colors?

Yes, you saw it, too, a fifth color (counting background) appeared when colors 1 and 2 were plotted close together. Anytime Color 2 is drawn to the immediate right of Color 1, a fifth color will appear unless the line drawn is horizontal. Such knowledge could be used to add another color in limited amounts, or it could be just another bug to look for, depending on your project.

So there you have it; Graphics 7-Plus in Basic. The program listing to set it up is but 48 short lines of Basic, and the method for using it is similar to the use of PLOT and DRAWTO in any other Basic program. A bonus is that FILL works in a more transparent

Table 1. Some Artifact Colors.

Setcolor	Color 1	Color 2	Color 3	Background
2,0,0	Yellow Green	Blue	White	Black
2,1,0	Yellow Green	Purple	Light Yellow	Brown Green
2,2,0	Yellow Brown	Purple	Light Yellow	Sienna
2,3,0	Orange Yellow	Blue Purple	Light Pink	Red Orange
2,4,0	Orange Yellow	Blue	Pink	Red
2,5,0	Yellow Orange	Blue	Pink	Purple
2,6,0	Yellow Green	Blue	Pink	Blue Purple
2,7,0	Yellow Green	Light Blue	White	Dark Blue
2,8,0	Light Green	Light Blue	White	Medium Blue
2,9,0	Medium Green	Light Blue	White	Blue Black
2,10,0	Green	Blue	Lime	Dark Grey
2,11,0	Green	Blue	Lime	Grey Green
2,12,0	Bright Green	Blue	Lime	Black Green
2,13,0	Light Blue	Blue	Light Green	Black Green
2,14,0	Yellow Green	Blue Purple	Light Yellow	Yellow Brown
2,15,0	Orange Green	Purple	Light Orange	Orange

manner even though the usual FILL idiosyncracies are a little more annoying with artifact colors.

This is but one more example of how the built-in flexibility of the Atari makes it possible to extend the application into areas beyond the limits of the original software design. If playing with this introduction to artifacting with Graphics 7-Plus has been enlightening, perhaps you may be able to use it. If the inherent bugs are too frustrating then you might try some language other than Basic that does not rely so heavily on the existing OS DRAW and FILL routines. Either way, happy artifacting.

Listing 6: Lines to Add to Listing 5 for FILL Subroutine.

```
399 REM ****FILL SUBROUTINE****
400 POKE 765,COLR:QX3=X:QY3=Y:Y=INT(QY
3+0.5)
405 IF Y>95.5 THEN QN=1:IF QY1<95.5 TH
EN QN=0:GOTO 435
410 IF Y<95.5 THEN QN=0:IF QY1>95.5 TH
EN QN=1:GOTO 435
415 POSITION X,Y-96*QN:XIO 18,#6,0,0,"
S:":QX1=X:QY1=Y
420 IF FLAG THEN FLAG=0:RETURN
425 XFLAG(0)=0:XFLAG(1)=0:X=QX3:Y=QY3
430 RETURN
435 QX2=X:QY2=Y:GOSUB 265:FLAG=1:GOSUB
 415
440 QN=1-QN:X=INT(X+0.5)
445 POKE 88,D88(QN):POKE 89,D89(QN)
450 GOSUB 265:X=INT(X+0.5)
455 Y=QY2:GOSUB 280
460 X=QX2:IF XFLAG(1-QM)=0 THEN QX(1-Q
M)=((95-QY1)*(X-QX1)/(Y-QY1))+QX1:XFLA
G(1-QM)=1
465 PLOT QX(QM),95-95*QN:DRAWTO QX(1-Q
M),95-95*QN
470 GOTO 415
```

Part III
Hardware and Software

Bits and Bytes

Question: How do you put four elephants in a Volkswagen?

Answer: Two in the front and two in the back.

It is a sad fact of life that every device yet made by mankind is subject to limits. We know that a Volkswagen is not designed to transport elephants. The Atari computer is a device with a lot of ability and thousands of potential applications, many of them probably beyond the expectations of the designers, but it does have limits.

The most fundamental limitation of the Atari as a general purpose computer concerns its memory constraints. In order to fit a great many capabilities in a small device at a reasonable price; decisions, compromises, and tradeoffs had to be made. This is especially true of the graphics capabilities of the Atari. Great flexibility, even at the cost of complexity, was one of the design objectives of the computer.

If you design a device so that a user has few options, it should be reasonably simple to operate. When you turn on a light at home, all you have to do is throw a switch, and perhaps occasionally change a light bulb. You do not have to be aware of hundreds of miles of light poles and wire cables, of transformer substations, and perhaps a complex nuclear power plant that make it possible for you to have light when you throw the switch.

The Atari is capable of operating on this level. You open the door on the top, plug in a *Missile Command* cartridge, plug a joystick into the front, press the Start button, and you are ready to defend your cities against nuclear attack. You need to know absolutely nothing about programming.

But the Atari also allows you to create your own programs, and every trick that is used in *Star Raiders* and *Eastern Front* is available to you as well. There is no way to give you so many options and so much power and keep the process as simple as throwing a switch.

It would be possible to simplify the process more than the Atari does by converting every instruction into plain English. Perhaps you could have a computer that could interpret a series of instructions like this:

- draw a man with blue eyes and blond hair
- make him a little taller
- give him a white shirt with a red and gold striped tie
- make the gold stripes thinner

This kind of graphics, while possible, would require a vast amount of development expense and lots of memory, and would lead to its own restrictions. In order to fit many capabilities into a small amount of memory, the Atari had to omit features that would make the graphics easier to use and understand. Instead,the programmer (that means you!), must gain a basic understanding of how the computer works.

The most basic element of information in any computer can be thought of as a two position switch. You can picture it as a light bulb that is either on or off, as a box that is empty or full, as an electrical circuit that is charged or not charged, or in any of several other ways. But the important thing is that there are only two possibilities. This fundamental unit of memory is called a *bit*. There are many bits of memory in your Atari.

(The 16K memory cartridge that came with your computer has 131,072 bits of memory. You can add two more such cartridges to an Atari 800, for a total of 393,216 bits. The operating system cartridge contains 81,920 more bits of memory, and your Basic cartridge another 65,536.)

Since any bit in memory can function as a two way switch, with possibilities as dramatic as turning the screen on or off, the computer can be very complex. In fact, a single bit in a fixed location can turn the screen display of *players* on or off. Much of the process of learning how to do fancy graphics on the Atari is learning how to find these special locations in memory and set the "switches" the way you want them.

In most of the memory locations in your Atari computer, a bit of memory is actually a tiny electrical circuit. This circuit is either charged with a voltage, or has no charge. As an easy way of referring to the state of one of these circuits, we use the numbers 0 and 1. If a bit of memory has a charge, it represents a "1" bit. If there is no charge, it represents a "0" bit. If you were to hook up a tiny light bulb to a single *memory cell* (one bit), it would actually glow when the bit represented a "1" and not glow when the bit was a "0". Many older computers had lights on the front to do just this; and the user could see what was in each memory location.

A group of 8 bits of computer memory organized into one unit is called a *byte*. The 8 bits of memory usually are in different locations in memory, but the computer is designed to treat them all as a single unit. The bits are organized so that position is significant. Each byte has a *bit pattern* that is a series of eight ones or zeroes representing the value of each bit.

A bit has two possible electrical states, on or off. How many does a byte have? There are 256 possible combinations of ones and zeroes in the eight bits that make up a byte. Just as a bit can represent either 0 or 1, a byte can represent a number from 0 to 255. (That is 256 numbers. Programmers start counting at zero.)

That may seem strange. Imagine a blackboard that was only wide enough for one digit to be written upon it. How many different numbers could it hold? The answer is 10, ranging from 0 to 9. If it had room for two digits, the possibilities would range from 0 to 99, giving 100 different possibilities. Since there are ten different possibilities for each digit, there are 10 times 10, or 100 different possibilities for 2 digits, 10 times 10 times 10 or 1000 different possibilities for 3 digits, and so on for numbers with more digits.

Since there are only two possibilities for a bit, there are only 2 times 2, or 4 possibilities for 2 bits. This means that the eight bits in a byte can represent 2 times 2 times 2 times 2 times 2 times 2 times 2 times 2, or 256 different numbers.

Bits and Bytes

There are thousands of different bytes in the memory of your Atari. Each of them is organized and assigned a number, which is called its *address*. Each byte has eight bits assigned to it, and every bit is a completely separate electrical circuit, or memory cell. You might wish to think of the memory of your computer as a very long street, with houses on only one side of the street, each house numbered in order, 0, 1, 2, 3, 4, etc. Each of these houses (a byte) has eight rooms (eight bits). In every room, the lights are either turned off or turned on (0 or 1).

You can locate any byte in memory to examine it by its number. The possible range of numbers with the current operating system is from 0 to 65,535. There is at least one device, the Axlon Ramdisk, that expands the Atari's memory to 262,144 bytes. These numbers may seem strange at first. Why not a "round" number like 100,000? The answer is that with two possibilities for each bit, the possible addresses must be a power of two.

How large a number can be represented by a certain number of bits? Here is a table for 0 to 18 bits. If you would like to check it out, just write down every possible combination and count them. For example, with three bits you could have 000, 001, 010, 011, 100, 101, 110, or 111; eight different possibilities.

Bits	Combinations
0	0
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256 (one byte)
9	512
10	1024 (also known as "1K")
11	2048 (2K)
12	4096 (4K)
13	8192 (8K)
14	16384 (16K)
15	32768 (32K)
16	65536 (Atari memory size — 2 bytes)
17	131072 (128K — Note this is NOT "131K")
18	256144 (The Atari can be expanded to this)

Because the processor in the Atari uses 16 bits to address memory, it can address 65,536 memory locations, numbered in order from 0 to 65,535.

How Your Atari Works

The "brain" of your Atari computer is a *microprocessor*, an integrated circuit. This processor is called a #6502, and its major function is to communicate with the memory in your computer. It receives instructions from memory and reads and writes instructions from and to memory and other devices (such as the cassette recorder, joysticks, the disk drive, and the screen) that appear to the processor (as if the other devices were also memory).

The 6502 starts at a location in memory and interprets the content of that first location as a command. From then on, the instructions tell it what to do with the other locations in memory and where to look next. The processor can only do this with the main memory, so every program must be in the random access memory at the time it is executed, or run.

Since we are limited to 65,536 locations in memory, and have to do everything in those locations, we must discover a way to store characters, displays, Basic variables and anything else as bytes of data.

Here are some ways information is stored:

Character strings. Each character (a letter, number, symbol, punctuation mark or space) is stored by representing that character as a number and storing the number into a byte. Since there are 256 different possible patterns in a byte, the Atari has 256 different characters. Each character is represented as a unique byte, and the computer looks up the number on a table in the operating system to determine what pattern to display on the screen.

Basic variables. Atari Basic is not limited to numbers between 0 and 65536, although it is limited to line numbers between 0 and 32767. This is because line numbers are stored in one bit less than two bytes, while Basic variables are stored as a group of 6 bytes. Basic encodes numbers in a way that makes them easy to work with, not in the manner we have been discussing.

Basic programs are stored in coded bytes, not as text. Each possible Basic statement type is given a unique number, and that number is stored as a single byte instead of as the letters that you typed into the computer. Whenever you list a program, those codes are translated back into letters that you can read. The details of this coding will be explained later in this book.

An important point to remember is that the 6502 processor sees data as bits and bytes. We think of data as characters, variables, or language keywords. Sections of memory are dedicated to different functions. For example, the operating system ROM holds 6502 machine language instructions, data for the display of characters on the screen, and text messages.

Physical Types of Memory

Computer memory is physically located on integrated circuits. While auxiliary "mass storage" is available on floppy diskettes or cassette tape, information must be loaded into the integrated circuit chips before the processor can work with it. It is important to know and understand the types of memory in order to take advantage of the special features of the Atari.

The best known type of integrated circuit is RAM, or "Random Access Memory." Random access means that any byte in that memory can be selected and accessed at any time. This is unlike a cassette tape, where you have to read through the tape from the beginning until you come to the information you want. However, Random Access Memory is not the best way to describe this memory, as other types of memory, including ROM, can also be accessed the same way. A better description is "Read-Write Memory," for the distinguishing feature of RAM is that information can be written to and read from the chip electronically in a few microseconds. The information can be and usually is changed frequently, often many times within the space of a single second.

The principal disadvantage of RAM is that information is stored only as long as electric power is continuously available to the chip. If the electric power is interrupted for even a fraction of a second, the information will be lost. The Atari power supply is designed to smooth out some minor power fluctuations, but it is still possible and even common for information to be lost due to momentary loss of power or voltage fluctuations. The chips were designed to store a lot of information in a small area. This required very low currents and close tolerances, which made the chips very sensitive.

But if RAM can only store information as long as power is available, how do we "wake up" the Atari computers? How can we store the information that the computer needs to get going, even when the power is turned off? We cannot read a program from disk or cassette. The computer needs a program to do that. Not too long ago, the operators of small computers had to physically load a *bootstrap* program into the computer by setting switches for each bit. ("Let's see now, up for 1 and down for 0, so 0010 0111 is down, down, up, down, down, up, up, up. Now we throw the switch to load that byte and go to the next address . . .") After a program of 60 or 70 bytes was loaded, that program could be run to load a longer monitor program from paper tape. If one bit was wrong, after throwing 500 switches, the program crashed and the operator had to start over. Fortunately, the Atari uses a better way.

Our second type of chip is called a "ROM", for "Read Only Memory". ROM is not affected when the power is turned off. The data in ROM is permanently burnt into the chip. (There are certain types of ROM which can be erased for special applications.) You cannot write anything to ROM because the hardware will reject it. Since ROM is always readable, it can be used to provide an initial program for the microprocessor to run. The operating system, the computer's main program, is stored in ROM. Remember ROM is used for programs that never need to be changed and those programs that can survive a power loss.

The Basic and Star Raiders cartridges contain programs written in machine language. Since the cartridge can survive a power loss when unplugged, it must be ROM. If you open up the cartridge, you will find two ROM chips. Atari cartridges have programs stored in ROM.

The ROM operating system is located inside the board that is plugged into the top of the Atari. This is a program the Atari uses to start up after a power loss. RAM cartridges with 8K or 16K of memory provide the read-write memory of the Atari.

An 8K RAM memory board has 8,192 bytes available and a 16K RAM board has 16.384 bytes available. The Atari can handle three 16K boards or 48K of read-write memory. There is 64K of total memory in the machine. The last 16K is split up into several other uses. One use is in the Operating System,ROM.

Think of the memory as a long thin line of bytes, each numbered individually. If we have a 16K board plugged into the Atari, and we write to any location from 0 to 16,383, we'll physically write something into the chips on that board. If we have a second 16K board, that will be the bytes from 16,384 to 32,767. A third board handles the next 16,384 bytes of memory. The last 16K of memory, (the total memory is in four 16K parts) from byte number 49,152 to number 65,535, is split up into other functions that do not require read-write memory. If you read location 60,000, you'll be reading from the Operating System ROM board. You cannot write to this.

Certain tools are provided in Atari Basic that are useful in gaining a better understanding of the memory. One of the tools is a function called FRE.

FRE is a way to determine "the number of remaining free bytes in RAM." When we type in a Basic program, we start to use RAM to store it. The storage in RAM is limited, so for every line typed in, there is less free RAM left. RAM has many purposes. Part of the operating system is stored there. Any tables that have to be saved must be stored in RAM since you cannot write to a location in ROM. Basic programs that you type in are stored in the remaining free RAM. FRE tells us how much RAM is left unassigned and is usable for the storage of Basic programs.

With only one 16K board in the Atari, we have only 16K of read-write memory. There will be considerably less space for storing Basic code. If we write to a byte that is not physically located on the memory board, our data will disappear and be lost. Extra memory is very handy, and this is why people are willing to spend extra money for it.

Two other Basic statements that will be valuable to us are PEEK and POKE. Peek gets a byte directly from a memory location that you specify, and puts it into a variable you designate. The number that will be shown will be the contents of that byte, a number from 0 to 255. POKE takes the number you give and puts it directly into the memory location that you indicate. The number that you poke should not be greater than 255 (a larger number will not work).

This is the format of PEEK: Variable=PEEK (address).

For example: to set variable A to the value of the contents of memory location 40,000, use:

```
A=PEEK (40000)
```

Let's say we wanted to dump a large section of memory to the printer. Here's a short program to do it.

```
100 START=40000
110 SEND=50000
200 FOR LOCATION=START TO SEND
210 LPRINT LOCATION, PEEK(LOCATION)
220 NEXT LOCATION
230 END
```

It's harmless to PEEK anywhere in memory. It will teach you a great deal about the machine. You can look at a Basic program in memory and find out exactly what it looks like to the computer as bytes.

POKE puts something into memory. Be cautious because if you randomly poke into memory you'll eventually rewrite a byte that the Atari needs to keep functioning. The result will be that the computer will "crash." You must then either press RESET or turn the power off and on again.

Type NEW to clear out any Basic program in memory, then type PRINT PEEK(8000). This will show you what memory location 8000 currently contains.

Type POKE 8000,100 to change the memory contents to 100. Now a PEEK at 8000 will return the number 100.

Many things can be done in the Atari with PEEKs and POKEs. Since everything the Atari does is based in memory, and PEEK and POKE are the only direct Basic memory modification statements, we'll be seeing a lot of them.

Let's review what we have covered so far. The Atari has up to 64K of memory, meaning there is around 64,000 individually numbered bytes. Each byte is composed of 8 bits. Bits can be either on or off, 1 or 0. A byte stores numbers through representing them with patterns in its internal bits. Since there are 256 possible combinations in 8 bits, a byte can store numbers from 0 to 255.

The microprocessor the Atari uses has 16 bits assigned to memory. The range of numbers 16 bits can represent is from 0 to 65,535. Because of this, the highest memory location the Atari can look at is also 65,535.

Memory is composed of RAM or ROM. The lower 40,000 or so locations of memory are RAM, depending on how much RAM memory you have installed. If you have installed just one 16K board, then the lower 16K of the 48K RAM area will be actual memory and the other 32K will be unusable. The upper 16K of memory is assigned to various purposes. Some of it is ROM, some is for other purposes.

Basic has several statements that allow us to directly work with memory. PEEK allows us to directly examine any byte. POKE allows us to directly modify a byte. FRE tells us how much free RAM we have available for storage purposes.

This should give you a pretty good overview of memory. We'll be dealing with memory throughout the book in more specific ways; for example, how is a string stored in memory? Let's move on now to the graphics section. In it we're going to deal mostly with memory and how to use it to generate graphics images.

Atari Music Composer

Karl Zinn & David Zinn

We've been using the Atari Music Composer in home education and some school situations. We would like to share our initial experience and preliminary ideas here, and suggest other things that could be done.

The manual for the Music Composer suggests it can be used to develop skills in listening, perception, music notation, composing (melodies, harmony and counterpoint), musical relationships, and building musical structures from simple parts. We found we could do all these things and more, always in a pleasant and rewarding educational environment. Nearly all of our trials were in a home setting; but some were in a summer class for 8 to 14 year-olds interested in using computers.

For those who know other music boards for small computers (ALF, MicroMusic, MicroTech, Symtek), this one is comparable with five important differences.

1) Nothing extra is needed. The circuitry is built into the Atari and the audio is amplified by the TV set (or monitor) which is used as the display device for the computer. You can also take the audio out of a 5-pin jack on the side of the Atari 800 to feed any other amplifier.

2) Most people will use it as given. Since the Composer software is in ROM it can't be changed. Programs can be written in Basic either to generate data files that can be read by the Composer, or to play the Composer's data files with other tonal characteristics.

3) Use is very straightforward, with most of the options so obvious that a manual is not needed. The user works through menu pages linked in a hierarchical structure, with clear mnemonics and using normal keys for insert, delete and cursor control.

4) The system protects rather well against common user errors. New users, without previous experience with computers, get melodies to play back about as they intended them, and are not likely to lose them accidentally.

5) The user has little or no control over tone quality, attack and decay, crescendo, and the like.

The basic building block is a musical phrase; up to ten can be stored in memory. Phrases are arranged in up to four voices, with dynamics, repetition and transposition specified in a list of statements which looks like a computer program. Indeed, the composition activity can be used to develop programming concepts such as sequencing and iteration. Building a melody and counterpoint from phrases is good practice in music education as well.

Karl Zinn, University of Michigan, Center for Research on Learning & Teaching, Ann Arbor, MI 48104.

David Zinn, Greenhills School, Ann Arbor, MI 48104.

Phrases, voices or an entire composition can be saved on tape or disk, and retrieved later, perhaps with new arrangements. We much prefer disk because it is faster, but the cassette was adequate when we put only one data file on the beginning of a tape. (You will have discovered this problem with positioning the tape when reading a file from the middle of a tape if you use cassette on the Atari. We have heard that this software problem in cassette control will be fixed by Atari in a future release of the operating system.)

We already said we hardly needed the manual. This should be true for almost any experienced computer user, and perhaps many novices. We find a five-minute demo to be enough to get anyone started; a few things may not be obvious, such as "FN" as the abbreviation for "File Name" in a prompt, and the prefix "D:" needed to specify that the file is to be retrieved from (or saved on) disk instead of cassette. But the manual is well-organized with clear descriptions and photos of the screen in various states. We recommend it to those who would rather learn systematically than by exploration. One part provides an overall description with things to do; another provides the file structure for those who wish to do things with Basic as well; it includes programs for listing files, composing music, and arranging harmony. A last part summarizes each of the commands.

SOFTWARE PROFILE

Name: Atari Music Composer

Type: Music

System: Atari 400 or 800, 8K

Format: Cartridge

Summary: Very well done at a simple price.

Price: $59.95

Manufacturer:
Atari, Inc.
1272 Borregas Ave.
Sunnyvale, CA 94086

We have many stories to tell about our use of the Music Composer, and plan to do so in a later article after we have experience with a greater variety of users and in other educational settings. Perhaps you can get an idea from these brief notes: Piano music entered into the Atari was played and displayed by the computer in a regular way which made obvious some syncopation which had been hard for the student to catch and perform otherwise. Some band music was entered so that the cornet player could practice (at home) with the other parts played by the Atari. A band part in the Atari was used as a model (and a metronome) for repeated practice of a difficult sequence, gradually coming up to the required speed. Music heard only on the air was entered and reviewed (and played for fun), exercising notation, interval recognition, note duration, time signature, key signature and other music components. The pleasure of this activity for kids contrasts with the reluctant response of some students to "dictation" exercises.

Music already stored in the Atari was modified in various ways (e.g., tempo and counterpoint) to change the style. Musical rounds and fugues were explored, pushing the complexity until the sounds were no longer pleasing to the arranger or composer. Timbre (tone quality) was explored by writing parts in unison and then transposing them to various partials (harmonics) one octave away, an octave and a fifth, two octaves, etc. Original compositions were developed by entering familiar melodies in up to ten phrases and rearranging them in interesting ways (such as those compositions of P.D.Q. Bach as discovered by Professor Peter Schickele!)

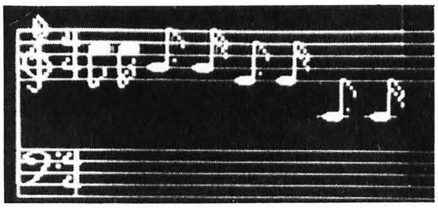
What we missed most while using the Atari Music Composer is a display of all four voices at once (as on a regular musical score or piano music). Sometimes it is difficult to find the part you wish to modify, since you can look at only one phrase at a time, and one measure in that phrase. Getting everything on the screen at once is a lot to ask of an 8K ROM application cartridge operating with an 8K RAM (yes, all these cartridges work on the 8K Atari 400 as well as our 48K 800) and displayed on an ordinary TV. If it weren't for the lack of resolution in TV rasters Atari might have avoided the problem of where to put the note stems by displaying each voice on a separate staff. Having a printout of the score would be really nice, and get around the TV display limitations.

At times we could enter music as chords instead of notes in separate voices. A good composer aid offer many options for entry of music. But being limited to one, entry in phrases and voices is the right one for this beginner's composer. Other advanced aids are also missing: tone quality, envelope (attack and decay), inversion, and other operations on musical patterns. We suspect that some of these can be done from Basic.

Although it is nice to be able to get all of the disk operating system from the Music Composer, working through it all to get a listing of what files are on the disk is a nuisance. One should be able to display the music files on the screen directly, and select one without the computer first erasing all the names. (It takes "D, <RETURN>. A, <RETURN>,RETURN" to get the directory on the screen. To get back requires a <RETURN>which erases the screen and then a "B, <RETURN>" to get back into the Music Composer. The new DOS 2.OS for the Atari simplifies this slightly (fewer returns are required) but one is still limited to what was designed into the Composer ROM.

In summary, although we could ask for more, what is provided was done very well for home education and recreational activities at a simple level. Clearly some people thought carefully about what should go into the Music Composer to make it helpful in music education. We hope others who find themselves in the position of advising computer companies will also help make the entertainment products better for education.

The Music Composer is available for $59.95 from Atari Inc., 1272 Borregas Ave., Sunnyvale, CA 94086. □



Hooking Up With CompuServe

Using Atari's Telelink Cartridge to Access CompuServe

Joining a club? Learn the ropes beforehand and it goes a great deal smoother. If you visit someone who has a computer, play a few games under your friend's supervision, and learn such simple tricks as the location of the ON/OFF switch and the use of the RETURN key, you are far ahead of the person who wins a computer as a doorprize at a convention, then takes it home and tries to use it with only the manual as a guide.

For the personal computer owner who has never used a large computer or a terminal before, seeking to connect to a timesharing service may seem to be an overwhelming task. All of a sudden you have to cope with learning how to use a new program, an RS-232 interface, a modem, and a telephone in cooperation with your computer. It can be quite frustrating, for if you overlook one switch setting or miss plugging in a single cable, the system won't work, and you might not be able to tell whether you made a mistake or whether one of the pieces of equipment is defective.

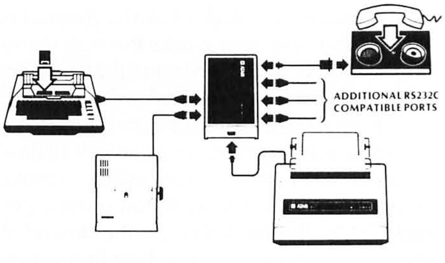
Failure-prone equipment in this type of situation is a disaster. For example, the Radio Shack TRS-80 Model I RS-232 board is notoriously hard to use, primarily because Tandy used a cheap connector to attach it. Some owners actually disconnect the board, clean the contacts, and reinstall it every time they use it. A new user with a bad connection might become so frustrated as to give up all hope of timesharing.

With the Atari Telelink cartridge, it took me several hours of work, accompanied by much frustration, to successfully hook up to CompuServe. I never did discover what I was doing wrong at first, but have come to the conclusion that the real problem was probably in our company telephone switchboard, not in the Atari equipment.

I hope that by a detailed sharing of the process that led to successful connection for me, I can make the same operation smoother for those of you who are considering timesharing.

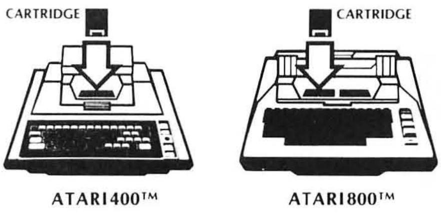
Equipment Required

The equipment I used was an Atari 800 Computer (the Atari 400 should work just as well), the Atari 850 Interface Module, the Atari 830 Acoustic Modem, the Atari Telelink I program cartridge, a telephone, and a Texas Instruments 99/4 Color Monitor. The only difference in my unsuccessful attempts was that I used a Leedex Video 100 black and white monitor instead of the color monitor. The difference was significant, not for the color, but because the TI monitor has a speaker, and you need the speaker to hear whether the cartridge loads properly. An ordinary television set should work as well, but I do not recommend any monitor or TV set without a working speaker. I did not have a printer, but it would have helped significantly.



SYSTEM DIAGRAM

The Atari Telelink I (The I probably implies that a II is coming!) program cartridge is a typical Atari cartridge. You load it by simply plugging it into the slot on the computer, a task that my six-year old son has mastered with the Star Raiders and Basketball cartridges. The Telelink I cartridge comes with a six-page foldout



Inserting the TELELINK cartridge

instruction brochure, a registration card, an application for an account with CompuServe, an instruction card for hooking up to CompuServe Information Service, and a sealed envelope containing a CompuServe user identification number and a secret password allowing you one hour of free access to the network.

The Atari 850 Interface includes a 102-page instruction manual that also covers the Atari 830 Modem. However, who is willing to read 102 pages of heavily technical material just to learn how to use an add on device on a computer system? Fortunately, you can use the manual strictly for reference, finding what you want in the table of contents.

Before you can use Telelink I to connect to CompuServe, you must have a local access telephone number. The card telling you how to access the timesharing service gives you Atari's toll free customer service number and tells you to call them for the access number closest to you. The customer service toll free number is very busy, and it took me about 20 calls over two days to get through. Once I did get connected, the representative gave me the names of cities in my area code with access numbers, and the telephone numbers.

Setting Up Your System

I will assume that you already know how to connect your Atari computer to a monitor or TV set, and only discuss the rest of the system. If you have a disk drive, disconnect it, as the cartridge is not set up to work with the disk operating system, and the two conflict.

If you place the Atari 850 Interface on the table in front of you so that the label faces you, you will see the following:

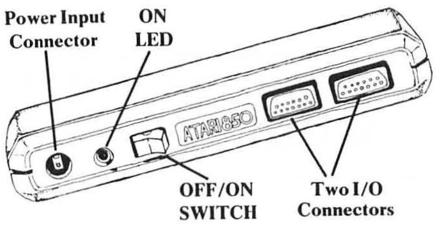
Plug the power supply into the connector on the left and connect it to a wall outlet. Connect the I/O cable from the computer to the leftmost one of the two I/O connectors. If you have the printer that uses the I/O connectors, connect the printer to the right front connector. If you have the Atari 825 printer, there is a connector for this on the right end of the interface module.

On the back of the interface module are four identical serial connectors, labeled from one to four. Plug the small connector on the cable that comes with the Atari 830 modem into connector one on the interface, directly behind the power connector. Your interface is now connected.

The connectors and switches on the Atari 830 modem are all on the same end, as follows:

onnect the other end of the modem cable from the expansion interface to the large connector on the modem. Then plug the power supply for the modem into the connector on the modem and into a wall outlet. If the power LED in the center of the modem should come on, set the originate answer switch to OFF.

Bring your telephone over to the computer. Place the handset so that the cord hangs over the end of the modem that contains the connectors and switches. This is also plainly marked on the label in the top center of the modem. Your system is now connected and ready to go.

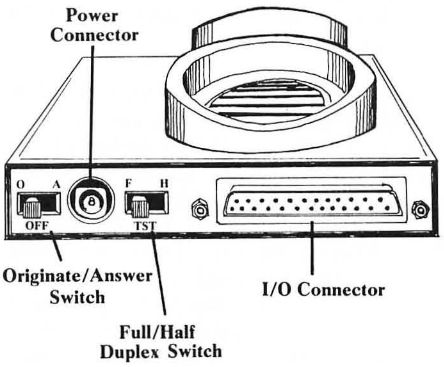


Making the Connection

Plug in your cartridge, and close the cartridge door. Turn on the television set or monitor. Then set the left switch on the modem to O (for originate) and the right switch to F (for full duplex). Both switches should be all the way to the left. The power LED on the modem should glow red. Next turn on the Atari 850 interface module using the switch on the front. The power LED next to this switch should come on.

After the rest of these connections are made, turn on the computer. If you turn on the computer before the interface, or have the disk drive connected, the program to operate the interface will not load properly. You should now hear a series of beeps from the television speaker to indicate that the program is loading. After the program has initialized, the words Telelink I will appear on the screen.

Now, dial the telephone access number for CompuServe that you obtained from Atari Customer Service. Unless the number is busy, it should ring a couple of times, then answer with a steady tone. When you get the tone, place the telephone handset in the cradle on top of the modem. Even before you finish placing the handset in the cradle, the two computers should recognize each other and the **READY** LED on the modem should come on.



Type **CONTROL C** on your keyboard. The TV screen should go blank, then CompuServe will print the message:

USER ID:

Respond by typing in the identification number in the envelope that came with your Telelink cartridge. Now CompuServe will print another prompt on your screen:

PASSWORD:

Type in your password, exactly as it is given in the envelope. The letters will not appear on the screen, so that you can keep your password secret if someone is watching. If you get it wrong, the computer will prompt you to try again.

CompuServe will now take a few seconds to log you in. It will recognize you as a new user and print a greeting message, plus give you instructions on using the system. It would be very helpful to have a printer turned on at this time to save the instructions for future reference. You will also be given an opportunity to open an account, either under Master Card or Visa or to be billed monthly, once your free hour is up.

Most of the time using the CompuServe network is as easy as reading the message on your screen, typing a number or a letter, and pressing **RETURN**. A few commands require you to type three or four letters, but these are explained.

My first time on the network, I read through the instructions for the various services, logged into the Atari Newsletter and sent a message to customer service, read several current stories from the New York Times, and looked through the other services. Then I typed EXIT and Compu-

Serve logged me off the system and told me that I had been connected for 29 minutes.

There are literally hundreds of other computer services that you can connect to with Telelink or similar systems from other manufacturers. There are other timesharing services, including The Source and universities such as the Dartmouth Time Sharing Service. There are many free message services all over the country. While some of them emphasize a particular computer system, most welcome all comers. You may want to try some of the services listed in the table. After you dial the number and get the tone, place your handset in the cradle and press **RETURN** a couple of times. The various systems should take you from that point. Please note that some of these numbers may be out of date when this article appears. Once you log onto several of them, you can usually find out about many more. Some of these numbers are only in operation after normal business hours for timesharing, as they are owned by businesses that use the lines during the day.

General Use		
CBBS	Pasadena CA	(213) 795-3788
(Community Bulletin	Akron OH	(216) 745-7855
Board Service)	Cambridge MA	(617) 864-3819
User Groups		
Forum 80 (TRS-80)	Chicago	(312) 269-8083
	Ft. Worth	(817) 923-0009
COMM 80 (OCTUG—TRS 80)	CA	(714) 526-3687
ABBS (Apple)	Seattle	(206) 244-5438
	New York	(212) 448-6576
PET BBS	Ypsilanti MI	(313) 484-0732
NORTHSTAR	Atlanta	(404) 939-1520
Interest Groups		
Genealogy	Fairfax VA	(703) 978-7561
Amateur Radio	Washington DC	(703) 281-2125
Commodities	Kansas City	(816) 931-3135
Avionics	Olathe KS	(913) 782-5115
Computer Stores		
Program Store	Washington DC	(202) 337-4694
Peripheral People	Seattle WA	(206) 723-DATA

For more information about the two most popular commercial timesharing networks, use these numbers. Thev are not numbers for computer access.

CompuServe	Columbus OH	(614) 457-8600
The Source	McLean VA	(703) 821-6660

The Telelink Program

As timesharing programs go, Telelink I is very limited, but it is also one of the easiest such programs to use. It is permanently set up for 300 baud (a rather slow rate of communication, especially when you are paying the phone bill), transmits even parity with one stop bit while receiving even parity or no parity, does not allow you to write files to disk, and has a fixed character set. If you try to access a computer system that does not accept any of these limitations, you will not be able to communicate. Actually, most timesharing systems are either set up this way or allow the user to specify his own configuration.

Telelink stores print characters in a buffer, so that you do not always have to wait for the printer to read the screen. You can turn the printer on and off from the program. It can communicate either Full Duplex (both computers sending messages at the same time) or Half Duplex (the two computers must take turns.)

Control characters that can be sent by Telelink I include TAB, ESCAPE, CONTROL A through CONTROL Z (including Linefeed, Bell, XON, and XOFF), RETURN, BACKSPACE, and RUB OUT.

The Atari 850 Interface

The 850 Interface module allows you to add four RS-232 serial ports and a parallel printer port to your Atari 800 or 400 computer. This allows you to connect printers, modems, and other standard peripherals to your computer. Although you would probably have to write the software yourself, you should be able to use it to connect lab equipment, a graphics tablet, a plotter, or other special purpose devices.

Atari does not currently offer any printer cables for use with the 850 Modem, except the one that comes with the 825 Printer. If you do not want to buy a $995 printer to get a $30 cable, you may be forced to create your own. To do that, you will need the part numbers and manufacturers of the appropriate connectors. The 25-pin parallel port uses an AMP connector, part number AMP 205-208-1. The 15-pin parallel ports use either AMP or Cannon connectors, part number AMP 205-206-1 or Cannon DB-15-P. The RS-232 serial ports use either the AMP 17-20096-1 or the Cannon DB 9-P connectors.

The Atari 830 Modem

This modem is a standard acoustic modem, very similar to the Novation CAT. By buying it from Atari, you get the Atari name on the label, and a cable that you know will connect to your interface, and Atari service. □

Build Your Own Light Pen

John Anderson

In this article, we'll take stock of a promising, yet somehow neglected input device for the Atari computer: the light pen. We will look at the capabilities of such a device, and review a pen available for the Atari as well as other machines. We shall go on to outline steps involved in the construction of an inexpensive but fully functional pen, using readily available parts.

If light pens don't sound to you like a topic that should necessarily elicit heated controversy or a complex and somewhat absurd tale, you are justified, but incorrect. Remember, you own an Atari, so anything is possible. Read on.

In the atmosphere of inspiration that couched the design of the Atari 400/800 computer, foresighted engineers built a great many capabilities directly into the hardware of the machine. Among these was the capability to support a light pen without the need for any additional controller boards. Even today, not too many other machines can make this claim. A light pen can be quite simply plugged into controller port 0, as if it were a paddle or joystick. It can be read straightforwardly with the statements PEEK(564) and PEEK(565). And that is all there is to it. That is, from an engineering point of view, you understand.

Those with machines of recent acquisition may not be aware that at one time Atari itself slated a light pen for production. It was to cost less than $100. In the second quarter of 1981, a products brochure that showed the device in use was released. It was a stubby, fat hunk of plastic with a tip switch on it. And what pretty multicolor pictures it supposedly drew.

Mail-order houses, as they are wont to do, accepted back orders on the Atari pen for some time. Though the decision to kill it was made over a year ago, the product was listed in a few retail rosters until only a few months ago.

At some point during its short development, a decision was made to pull the pen. The reasons for this remain somewhat vague. Some have suggested that the tip switch was flaky, making the device unreliable.

Another explanation I have heard from more than one reliable source goes like this: The Atari is designed as the machine for *everybody*, including novices and kids. Marketing was skitterish about the idea of a tiny kid fooling around a TV tube with a big pointy stick. One false move and gazonga: Mommy finds Billy on the living room floor, a victim of implosion! "Think of the lawsuits," said the legal department. "Pull the pen," said marketing.

Stop laughing. This may or may not have been the last straw concerning the Atari light pen. Whether it was or not, the pen was pulled from production very swiftly, and it is unlikely the decision will ever be reversed. A few did manage to get off the assembly line, however, and the few people who own them quite properly regard them as collector's items.

Hobbyists like myself, who have read about the capabilities of light pens and know also of the built-in pen capabilities of Atari machines, awaited the appearance of Atari-compatible light pens from other sources. Surprisingly, at least to me, no cheap pen has become available in the ensuing time. It is too bad, really. The peripherals can do a lot to make a microcomputer friendlier.

Just how can they do this? Kind of you to ask. First, let's find out what they do.

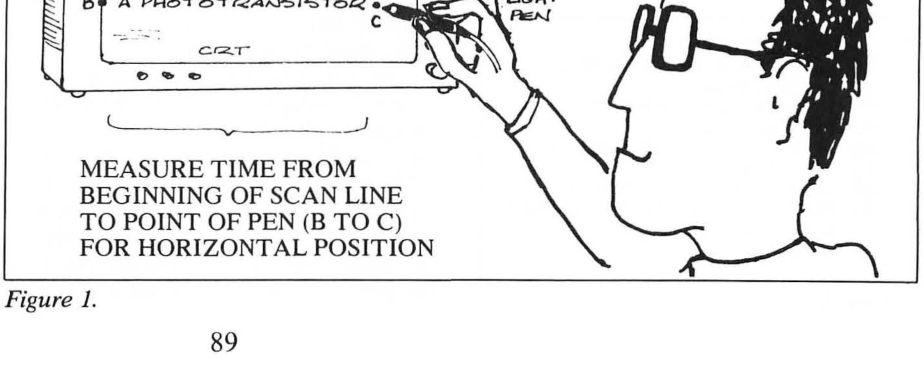
Light On The Subject

A light pen, when touched to or aimed closely at a connected monitor or TV screen, will allow the computer to determine where on that screen the pen is aimed. The driver program may subsequently take that information and do various things with it, but the job of the pen itself is quite simply to make a time measurement, which will be translated into x and y coordinates representative of a position on the CRT.

The capability may seem remarkable, and it is, though a simple explanation of how it works may dispel some of the awe. You may be aware that a TV or raster monitor typically *refreshes* at a rate of 60 frames per second. That is to say the electron gun or guns draw 60 pictures on the screen in one second. But it is impossible to draw an entire picture at once. Rather, the picture is drawn by the *scan line*, starting in the upper left-hand corner, moving to the right. When a line is completed, work begins on the next line. The Atari standard is 192 scan lines per frame. (An excellent explanation of this mechanism was provided by David Small in the June and July 1981 issues of *Creative*.)

Now let's imagine we have a special kind of transistor: one that is sensitive to light. We have hooked this transistor to our Atari, and aimed it at a point on the screen. By noting when a scan goes by and measuring the interval between scan lines or entire screen refreshes, we can get a good idea where the phototransistor is pointed on the video screen. The pen then allows us, through software, to generate x and y vectors corresponding to a point on the screen, which we may then use to draw pictures, make a choice from a menu of alternatives, or answer questions put to us by a program. Figure 1 is a simplified diagram of the process.

As opposed to input via the keyboard or even a paddle or joystick, a light pen can be a dramatically friendly peripheral. Imagine needing merely to point the device at your choice on the screen, in order to make that choice. Or to draw a picture on your CRT as straightforwardly as you might use a crayon on a piece of paper. These are the kinds of



MEASURE TIME FROM BEGINNING OF REFRESH TO CURRENT SCAN (A TO B) FOR VERTICAL POSITION

MEASURE TIME FROM BEGINNING OF SCAN LINE TO POINT OF PEN (B TO C) FOR HORIZONTAL POSITION

Figure 1.

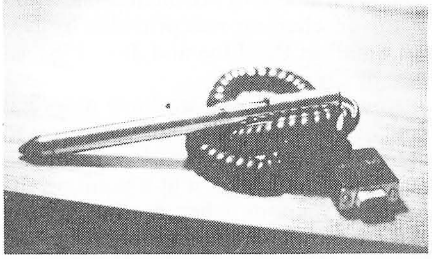
John Anderson is an associate editor for *Creative Computing* magazine.

possibilities a light pen affords.

By the way, you would have to work extremely hard to push a light pen *through* a CRT. It just isn't something you could do without extreme effort, assuming you could do it at all.

Mightier Than The Sword

Soon after the Atari pen bit the dust, a third-party pen for the Atari appeared from Symtec Corporation. This pen is about the most professional you can find for any machine. It is, in fact, an adaptation of the same model used in professional mini and mainframe operations. Its barrel is of heavy, extruded aluminum, with a coiled telephone handset wire leading to an Amphenol connector. It includes a sensitivity trimmer adjustment. Everything about the Symtec pen is top of the line, including the $150 price tag.



Symtec Light Pen.

Figure 2 provides an example of the drawing capabilities of the Symtec pen. The software driver I used to create the caricature (portrait) of our fearless leader, Mr. Ahl, appears as Figure 3. In ten lines, the code evidences how elementary a driver can be. This is an obvious benefit of the fact that so much of the work is already done in hardware.

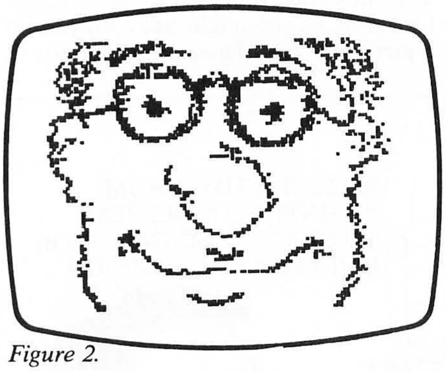


Figure 2.

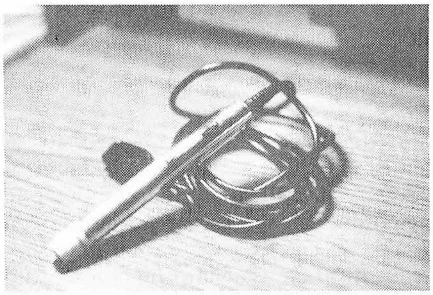
```
 10 GRAPHICS 7+16
 20 SETCOLOR 4,0,14:COLOR 3
 30 X=PEEK(564)
 40 IF X<70 THEN X=X+230
 50 Y=PEEK(565)
 60 IF Y<17 OR Y>112 THEN 50
 70 X=X-75:Y=Y-14
 80 IF X<0 OR X>159 THEN 30
 90 TRAP 30:IF STICK(0)=15 THEN
    PLOT X,Y
100 GOTO 30
```

Figure 3.

If you wish to endow your Atari with professional light pen capability, the Symtec pen is literally without rival on the market. The pen is also available for the Apple, IBM PC, and VIC-20 machines. For more information, contact Symtec, 15933 West 8 Mile, Detroit, MI 48235. (313) 272-2952.

Penlight Light Pen

Of course, many Atari hobbyists will be unable to budget that kind of money for a light pen purchase. I believe the market exists for an inexpensive pen, but no company has yet stepped forward with such a product. Other inexpensive pens, for machines such as the Apple and TRS-80, can be modified for use with the Atari. I reasoned, however, that it wouldn't entail very much more work to start from scratch. It would also be much cheaper.



Home Brew Light Pen.

The result: for a couple of hours work and about $10 worth of hardware, you can put a homemade Atari light pen to work with your system. While it will have neither the accuracy nor the feel of the Symtec pen, it will be perfectly serviceable for many applications, and loads of fun to play with. It is also easy to make. So let's make one!

First, you've got to stock some parts. Get down to the nearest Radio Shack, and pick up the following: one phototransistor, model number 276-130, 89 cents: ½ watt 100K ohm resistor, model number 271-045, 19 cents for two: penlight, model number 61-2626, $1.99.

You will also need a few other pieces of paraphernalia. These include: DE-9 connector plug for the controller port on the Atari, and five-conductor shielded cable (you cannot use an existing Atari joystick, as it lacks necessary pin-outs); a couple of feet of insulated bell or stranded wire; and the plastic top to a Bic pen. You may also want a grommet or strain relief for the pen top.

For tools, you'll need this array: low wattage soldering iron and solder; wire cutters (needlenose pliers are handy too); X-acto or razor knife; scissors or reamer; small flat blade and Philips screwdrivers; long stick pin or safety pin; and insulating electrical tape.

Got these things together? Let's get going. First, unscrew the cap on the penlight, and disassemble the light bulb and bayonet assembly from the white plastic pen tip. Next, gently press the switch assembly down through the barrel of the pen with the Philips screwdriver. We don't want a penlight anymore, and we need all the real estate inside it in order to convert.

The cable we connect will feed through the hole where the on/off switch used to reside. You will pop the switch out through the open side of the barrel, along with two springs and a black plastic retaining collar. When these things have been pushed out, the barrel will be empty, and that's the way we want it.

Using a closed pair of scissors or a reamer, enlarge the switch hole on the metal barrel top until it accommodates the wire, grommet, or strain relief on the connector wire you have chosen. When this is accomplished, push the pen barrel onto the wire (it would be embarrassing to construct the entire pen, then discover you left the barrel aside, and have to disassemble all your work to fit it on).

Take the phototransistor, and hold it so that the bottom is facing you. Turn it

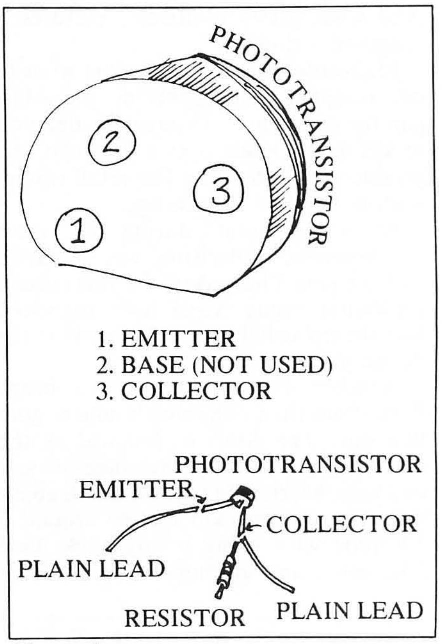


Figure 4.

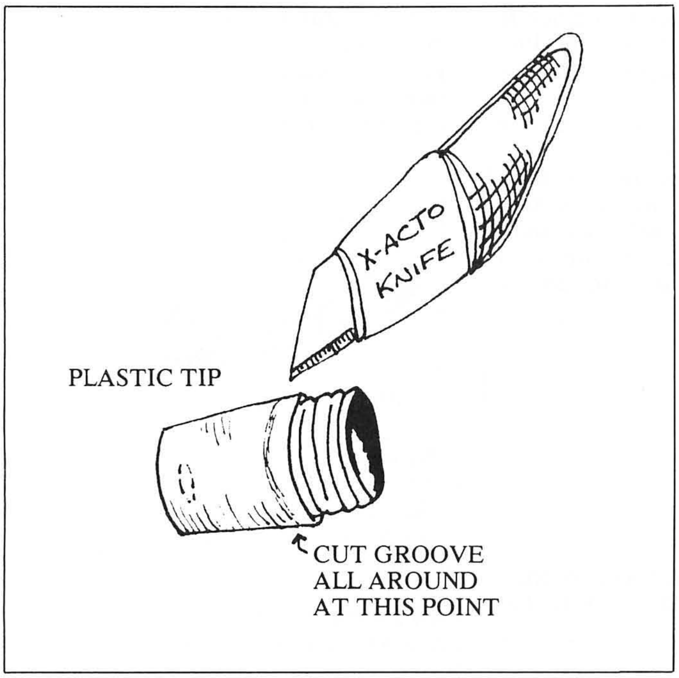


Figure 5.

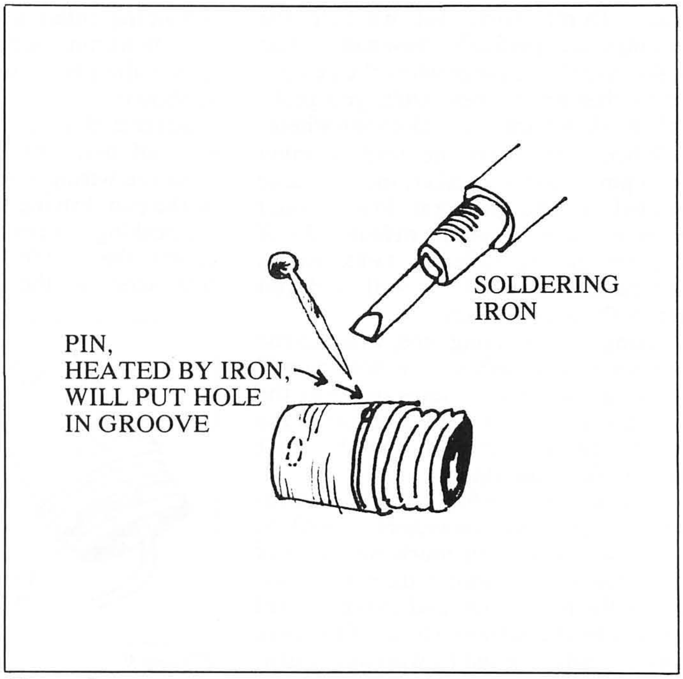


Figure 6.

until it is oriented along the lines of the diagram presented as Figure 4. This will indicate the positions of collector, base, and emitter leads of the component. You can clip the base lead short, as we will not be making use of it.

Solder directly to the collector lead one 100K resistor, along with a plain lead about four or five inches long, as indicated in Figure 4. Solder another lead of about the same length to the emitter lead, also as indicated. Don't use a high wattage iron or apply heat for too long, as you run the risk of blowing the transistor.

Using the X-acto knife, cut all the way around the plastic tip of the pen light, at a distance of about $^{1}/_{8}$ of an inch up from the threaded side, as indicated in Figure 5. Run the blade around the plastic tip repeatedly, until a rudimentary trench begins to appear. Once it does, use the flat blade screwdriver to widen and deepen the groove. This groove will hold the touch ring, which we shall use as the switch on our pen, in place.

Next, using the stick pin or an open safety pin, you will put a hole in the groove. Place the end of the pin in the groove, then put the tip of the soldering iron on the pin. Grasp the pin with the pliers or far enough back to avoid burning yourself. The plastic will melt only around the pin, and you'll have a clean hole through the pen cap. Work the hole out to about the diameter of a pencil lead. The touch ring wire will have to fit out and back into the pen through this hole. Figure 6 will help you gain a clear idea of what you're trying to do.

Figure 7 indicates the manner of construction of the touch ring. Strip a five inch or so length of wire entirely. If it is stranded as opposed to solid wire, make sure that you have twisted it together thoroughly, or it will unravel while you are threading it into the pen tip. The wire will loop all the way around the pen tip, into the groove hole, and should be tightly twisted to itself on the inside.

We are now ready to wire up the pen. Figure 8 provides a wiring diagram for connection to controller port 0. We shall be using the analog reading of Paddle (0) to tell us whether the touch ring is open or closed. The ground, pin 8, and the Paddle (0) hot lead, pin 9, form the touch ring circuit. As it turns out, this is an extremely convenient manner in which to activate and deactivate the pen. The resistor is connected between the collector and +5 volts, which is pin 7. The collector is also connected directly to pin 6, which is the hot pen lead. The emitter attaches to ground, which as stated, is pin 8 on the controller plug.

After the connectors have been soldered together with their respective leads, a test of the pen is in order, to make sure everything will be working when it is assembled. Plug the pen in, boot Atari Basic, and type the following:

```
10 SETCOLOR 2,0,14:SETCOLOR
1,0,0:?
PEEK(564),PEEK(565),
PADDLE(0):GOTO 10
```

Upon running the program, hold the phototransistor up to different points on the screen, and ascertain that you are getting different readings for each po-

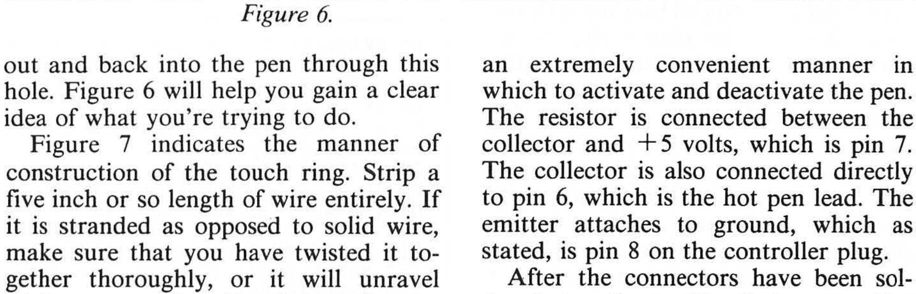

"LASSO" TIP WITH BARE WIRE
PUSH WIRE ENDS THROUGH HOLE
PULL TIGHT, BRAID WIRE TOGETHER

Figure 7.

sition. Don't worry yet whether the readings are perfectly reasonable. Just make sure they change when the pen position changes. If they don't, you probably made a wiring mistake somewhere.

When you touch the leads coming from pins 8 and 9 together, the last value printed in the program loop should move well down from its default, 228. If you are getting different PEEK values and paddle values, all is well, and you are in the home stretch.

Using the insulating tape, wrap up the pen wiring assembly so that nothing will short out when it is squeezed into the pen barrel. There is plenty of room in the pen for the assembly, so you shouldn't have to force anything.

Next solder two four- or five-inch insulated leads to connectors 8 and 9, which will detect our touch ring. One of these leads will connect directly to the tail of the touch ring, and the other will ground to the exterior barrel of the pen. This is easily effected by wrapping a generous length of stripped lead through the square hole in the plastic tip, as indicated in Figure 9. Then, when the plastic tip is screwed on, a good ground connection will be made via friction fit.

It is imperative that the connection to the touch ring itself be well insulated—your electrical tape will come in handy again here. Make sure no bare wire is left to accidentally short the switch. That way it will only close when your finger shorts it.

We're almost done. Bet you have been wondering what our Bic pen top is for. Well, now we need it. Cut off the tip and the bottom with the X-acto knife, as shown in Figure 10, so just about a half inch from near the top is left. This remaining collar will act as a guide for the phototransistor in the pen tip. Press it into the plastic tip, tapered side first, as shown.

After making a final inspection to ensure all bare wire has been insulated, push the wiring assembly into the barrel of the pen, leaving just the phototransistor peeking out about a half inch, and of course the switch leads and tip. Carefully screw on the tip, making sure that the phototransistor is seated well in the pen collar, and that a satisfactory ground connection is being made between the lead looped outside the plastic cap and the barrel of the pen. And that's it.

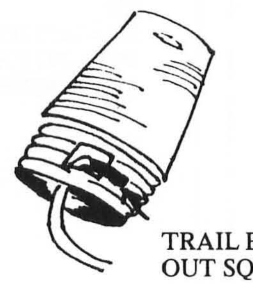


Figure 9.

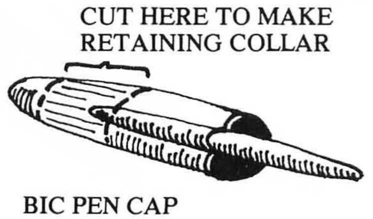
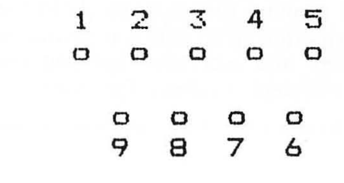


Figure 10.

There are some hardware 800 models which cause the light pen to be read from port 4 on the 400. If you have a 400, plug the pen into port 4 and substitute PADDLE (6) for all references to PADDLE (0) in the demo programs.

```
10 GRAPHICS 7:SETCOLOR 1,0,0:SETCOLOR 2,12,14:SETCOLOR 4,0,14:COLOR 1
20 POKE 752,1:? "TO DRAW, TOUCH THE PEN TO THE SCREEN,"
30 ? "THEN TOUCH AND RELEASE THE RING."
40 IF PADDLE(0)=228 THEN 40
50 X=PEEK(564):IF X<50 THEN X=X+230
60 Y=PEEK(565)
70 X=(X-95):Y=(Y-14)
80 TRAP 50:IF PADDLE(0)<228 THEN PLOT X,Y
90 IF PADDLE(0)<228 THEN 90
100 ? :? :? "TOUCH THIS BAR TO REVERSE COLOR, AND"
110 ? "BELOW THIS BAR TO ERASE."
120 X=PEEK(564):IF X<50 THEN X=X+230
130 Y=PEEK(565)
140 X=(X-95):Y=(Y-14)
141 IF Y>96 THEN 10
142 IF Y>83 AND Y<96 THEN 300
150 TRAP 120:IF PADDLE(0)<228 THEN DRAWTO X,Y
160 IF PADDLE(0)<228 THEN 160
170 GOTO 120
300 Y=0:SETCOLOR 4,0,0:? :? :? "TOUCH THE RING TO CONTINUE..."
310 IF PADDLE(0)=228 THEN 310
320 X=51:Y=0:SETCOLOR 4,0,15:GOTO 20
```

Figure 11.



NOTE: THIS IS THE PLUG.
THE JACK WIRES UP IN
MIRROR IMAGE.
6 - (PENHOT) - DIRECTLY TO COLLECTOR
7 - (+5v) - TO RESISTOR AND TOUCH RING
8 - (GROUND) - TO EMITTER
9 - (PADDLE O HOT) - TO PEN BODY

Figure 8.

```
10 GRAPHICS 0:SETCOLOR 2,0,0:SETCOLOR 4,0,0
20 ? :? :? :? :? "Question 1."
30 ? :? "How many zweckas does it take to fill"
40 ? "a quackenbush?"
50 ? :? :?
60 ? "□ONE"
70 ?
80 ? "□TWELVE"
90 ?
100 ? "□HUNDREDS AND HUNDREDS"
110 ?
120 ? "□WHO CARES ABOUT QUACKENBUSHES?"
130 Y=PEEK(565)
140 POSITION 2,22:IF Y>60 AND Y<64 AND PADDLE(0)<228 THEN ? "You must
    have a tiny quackenbush! "
150 POSITION 2,22:IF Y>66 AND Y<70 AND PADDLE(0)<228 THEN ? "No, but
    there are in a dozen.       "
160 POSITION 2,22:IF Y>76 AND Y<80 AND PADDLE(0)<228 THEN ? "You bet
    it does, buddy.             "
170 POSITION 2,22:IF Y>82 AND Y<86 AND PADDLE(0)<228 THEN ? "That's
    the wrong attitude to have."
180 GOTO 130
```

Figure 12.

Conduct another test, identical to the earlier one. If results are unsatisfactory, you'll have to undo things and find out where you went wrong. If you are having trouble activating the touch ring, try wetting your finger before you dismantle anything. Because we are reading the resistance between the ring and the barrel of the pen, a dry finger can sometimes be the culprit.

You should now have a relatively neat looking as well as functional light pen, that passes the one-line software test with flying colors. The time has come to begin refining that software dramatically.

I will provide two starting points. Figure 11 is a drawing program, which will give you an idea of how good (or bad) the pen is at locating itself. I built three pens, and the calibration seemed pretty consistent among them. Of course your monitor will have much to do with pen calibration.

The first place to look is line 70. Values in this line should be altered until the plot occurs right underneath the pen tip. If the left side of the screen reads okay but the right half is out, you may have to fiddle with the value in line 50. Don't get nervous. For most folks, the values shown in the program will be pretty close to perfect.

You will quickly see that the pen is much more accurate at vertical measurement than at horizontal. This is probably its biggest shortcoming, though it has others. For one, the screen must be extremely bright to get a good reading. For this reason I have included an option to reverse color, which is chosen by pointing the pen to the text window and touching the ring. To erase, move the pen below the bottom edge of the text window.

Figure 12 is a simple menu selection program to give you an idea of the convenience of a light pen for varied information input. The pen you have built is more than accurate enough to support a function such as this. The squares in the listing are obtained by pressing the Atari key, followed by a space.

Needless to say, these examples are presented just to get you started. Your imagination can take it from here. So there you have it. You need never be stymied again when people ask you about the light pen capabilities of the Atari machine. In fact, they may be sorry they asked! □

Atari Silencer

John Anderson

It is commonly known that, in addition to the capability of driving sound through a television or monitor speaker, the ATARI has an onboard speaker, similar to the Apple II. This speaker can and does serve in a number of capacities, not the least of which is to sound a prompt or signal tone, to flag a specific mode or indicator.

Users of the 410 program recorder are familiar with the record and play tones sounded as an indicator before data input or output to tape. All users should be familiar with the chirp of keyboard feedback. This feature lends a surer "feel" to the keyboard than that found with other computers.

These features are, essentially, well-designed and helpful. However, I've discovered that there are times I wish I could fit a silencer onto my ATARI 800 Late night editing sessions or programming when my roommate is trying to catch forty winks have caused friction. Certain programs I use very frequently, like the *ATARI Word Processor,* seem to exploit the feature to a point beyond distraction. These features are helpful in a noisy office environment, but seem a bit heavy-handed in a quiet work area at home, the most common environment for the ATARI. I nearly discontinued exploration of a hi-res adventure because the program continually prompted for pressing RETURN with a long, shrill "blat" — shades of operant conditioning! Is it too much to ask to be able to turn the thing on and off at will?

What could be simpler than the installation of a single pole, single throw switch to cut out the speaker when desirable? A "take-aparter" since earliest childhood, I had already

Figure 1

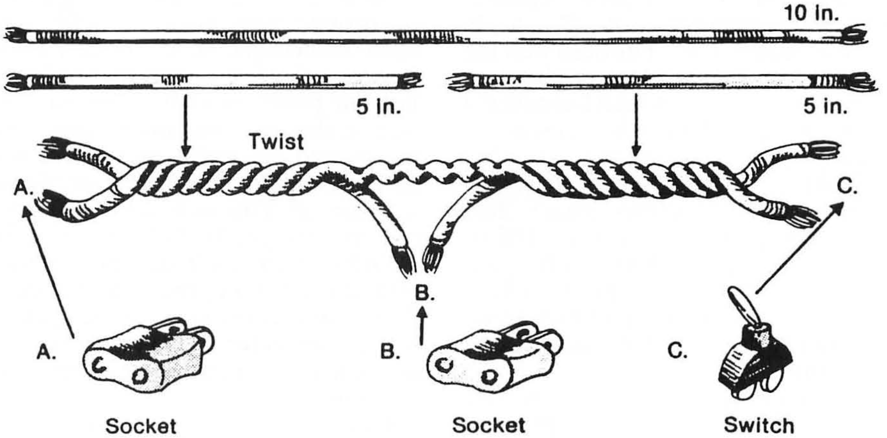
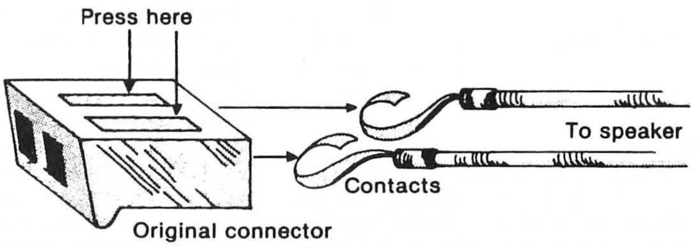


Figure 2



Illustrations courtesy of Softside magazine.

John Anderson is an associate editor for *Creative Computing* **magazine.**

Figure 3

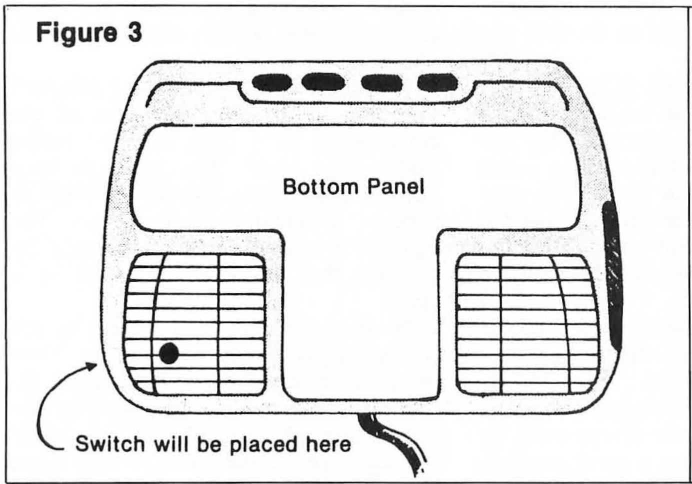
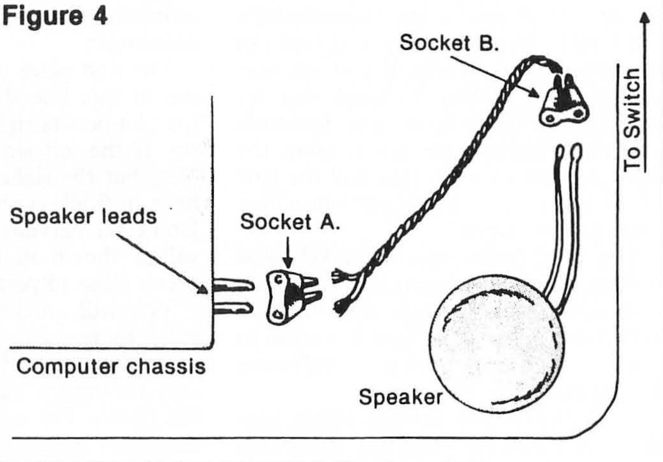


Figure 4



snooped around a bit inside the ATARI, and knew how easy it really would be. But, I still had a problem. The mere thought of snipping wires or drilling holes in my pristine machine made the hairs on the back of my neck stand on end. Also, though my warranty had long since expired, I wasn't happy with the idea of doing anything that couldn't be undone. Service people can be put off quickly when they see user modifications. I determined, rather wistfully, that I could live with the buzzers.

Then, while staring at all the little packages hanging on the wall of a nearby Radio Shack, I made a fascinating discovery — I saw a product called "two prong connectors," catalog number 274-342—$2.49 for a package of six. I noticed that the fit would be quite close to the connector used on the ATARI speaker. I then noticed "SPST micro miniature toggle switch," catalog number 275-624 — $1.59. Smaller than the smallest switch Radio Shack had stocked previously — it occurred to me that it would fit between the vent slots on the bottom of the ATARI. I suddenly envisioned a switch modification that was totally, and easily, reversible.

The modification was a complete success. Now that I can toggle the speaker off, I realize it's something I should have done long ago. In case I need to bring the computer in for service, the modification can be slipped out in under five minutes.

The Project

If you wish to modify your ATARI, you will need, in addition to the products listed above, about two feet of bell or other light wire, a flat blade and Phillips screwdriver, soldering iron and solder, and a bit of tape.

Snip the wire into two ten inch lengths. Then, take one of the wires and snip it into two five inch lengths. Strip a quarter inch of insulation off the ends of all the leads. Twist the shorter wires onto the longer wire in the manner indicated in Figure 1. This will make the modification easier to slip in and out later. Next, solder two connectors and the switch to the wires as indicated in the diagram. Unscrew all collars around the neck of the switch. Notice you are using only the *socket* connectors, not the *plug* connectors. Leftovers can be saved for another project.

Now you are ready to begin the operation. Flip your ATARI over onto something soft, like a pillow. Unscrew the five screws that hold the bottom panel, and lift it toward you. Notice that the controller ports must be cleared in order to remove the panel. Can you believe how small that speaker is? You now know another reason why you're lucky to own an ATARI. You don't depend on that little thing for all your sound effects. To disconnect the speaker, pull gently on the connector. Once the speaker is disconnected, remove it from the machine.

Orient the connector so that it matches the view in Figure 2. Using a screwdriver or toothpick, press down on the silver tongue on the top of the plastic connector, as you gently pull the wire from the side. **Don't force anything!** When you've pressed the tongue down far enough, the contact will slide right out. Pull both contacts out of the plastic container.

Next, take the bottom panel you removed earlier and hold it so that the vents are at the bottom, as shown in Figure 3. You will mount the switch in the left-hand vent, where there is room to spare, and nothing nearby that might get shorted out. Insert a flat blade screwdriver between the two vent slots where the switch will be mounted. (It's a good idea to stay over to the left — this will make the switch easier to reach.) Gently twist the screwdriver to spread the slot, then press the neck of the switch through. The plastic will have to bend a bit to accommodate the switch. Put on a washer, then screw on the lock nut to fasten the switch in place.

The final installation will be facilitated by repositioning the back panel so that the computer looks like an open valise. This way, the wire between switch and speaker will not be stretched. First, press the speaker contacts into the middle connector, as indicated in Figure 4. The speaker can now be repositioned in its place. Gently connect the far socket to the speaker leads from which you removed the original connector. Spreading them a bit may insure a tight fit. Finally, tuck the wire away under the keyboard post and away from boards and the speaker itself. There's enough room on that side of the computer to insure that the modification will not interfere with any other hardware.

You may wish to tape the original connector to the wire itself. Then, should you wish to remove the modification, the original connector will be right where you left it.

Screw the back panel on, plug things back in, and run a test. You can easily use the keyboard REPEAT function in memo pad mode to do this.

Listen. You can almost hear a pin drop!

A New Basic For the Atari — Basic A+

Mike Dunn

Atari Basic was originally written by Optimized Systems Software, at the time a division of Shepardson Microsystems. There have been two patterns of Basic in microcomputers originating from Hewlett-Packard and Digital Equipment Corp. Bucking the trend toward the DEC orientated Microsoft dialect, Atari Basic was patterned after the Hewlett-Packard model, as were NorthStar and Cromemco Basics. Contained in an 8K ROM cartridge, the Atari now has some new Basic's available. One is Microsoft Basic and the other is an extension of the original by Opimized Systems Software, now an independent company. This article is about OSS's new Basic A+ and Disk operating system, OS/A+, both written by the same individuals who wrote the original Atari Basic.

The new Basic adds 5K of new features to the original Basic, with a total of 43 new statements and functions. Basic A+ is a structured Basic and has many user-oriented features that make your Atari easier to use. The statements IF..THAN..ENDIF and WHILE..ENDWHILE help you write programs in a structured style, and the listings produced will automatically be displayed with the proper indents. Nested loops are easily followed. Lower case and inverse characters can be used for commands, but LIST in upper case.

For business use an extensive PRINT USING capability as well as a TAB functions are included. There are commands for developing fixed length records for random access files, and an INPUT".." statement that allows you to specify the prompt displayed; it is also self-trapping so it will automatically reprompt if the input given is in error. Other improvements include the ability to use subscript with INPUT and READ statements, and added string handling functions such as string concatenation using "," and new commands such as FIND that search for a particular substring. Aiding in program development are TRACE functions, SET and SYS commands that allow easy changing of default values instead of POKEs, meaningful error messages instead of numbers, and the ability to call up the disk directory easily in Basic, "IF ERR" instructions can be used to test errors and direct the program flow. You can DELete any lines between two numbers and easily move back and forth between Basic and Assembly Language with several special commands. Two-byte words are directly accessible with DPEEK and DPOKE functions. All the variables used in the program can be instantly listed, and all the usual Disk commands are directly available.

The most exciting feature is the extensive set of Player-Missile Graphic commands that make using these special functions as easy to use as PLOT and DRAWTO with regular graphics. The Atari comes with 9 graphics modes, with various amounts of resolution and numbers of colors available. Independent of this system called the playfield, is a system of Graphics called Player-Missile Graphics, that allow incredible feats such as overlays with user defined priorities, machine speed graphics, collision registers, shape tables. These special effects that can be seen in Atari games such as Star Raiders. With Atari Basic, using PEEKs and POKEs and machine language subroutines, these special hardware registers can be accessed, but with Basic A+, simple Basic commands allow you to use this system with ease. There are 14 of these commands available for use, as well as new integrated joystick commands that simplify Joystick use. These joystick commands also make the movements "silky" in feel and much smoother than found in even the ROM Atari Games.

Program number 1 listed in this article is similar to the one by Chris Crawford of Atari, Inc., published in January 81 "Compute" magazine. In that article, Chris explained how to use Player — Missile Graphics by defining a space ship and moving it around the screen with a joystick. The shape table made first by a binary image, is then converted to Hex and then to decimal. The Shooting of missiles is also demonstrated in this article. Some important features illustrated in the program listed here include the format that the listing is printed in, as this is the way the program actually appears on the screen. The commands that begin "PM..." are some of the special Player-Missile graphic commands of Basic A+. Line 100 begins the WHILE... ENDWHILE loop and will execute as long as WHILE is non-zero. Also note the simple commands to use the joystick. Line 110 moves the spaceship around the screen. The second program draws the same spaceship, then shoots missiles. The BUMP command on line 260 access the collision registers, as many lines between IF...ENDIF as you wish.

Basic A+ comes on a disk, and can be easily changed by a FIXER program if "bugs" are discovered. In the future, many other useful items are planned to further enhance Basic A+. An APPLE version is also planned, compatible with the Atari version, so programs from one will work on the other (except, of course, hardware dependent features such as Player - Missile Graphics).

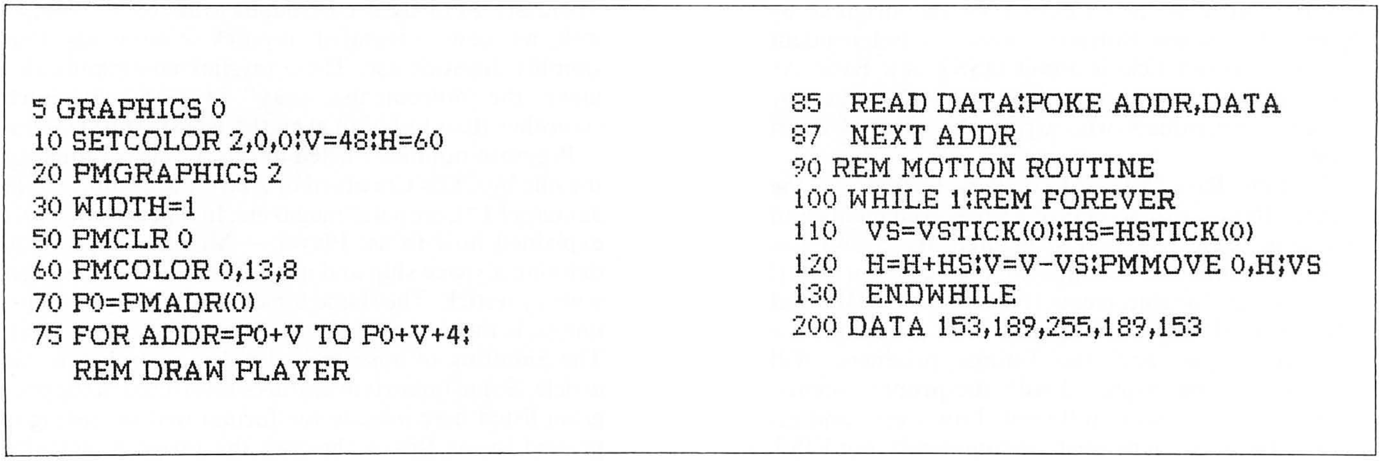
A new disk operating system called OS/A+ has been also released as an independent but integrated part of the package. It includes an Assembler, as well as many utilities in a command oriented format rather than the menu driven format of the Atari DOS. BATCH processing data via EXECUTE files makes the Disk system more powerful than ever before, as it allows the user to string together a series of programs, and a STARTUP. EXC command allows you to specify any file to be run on booting the disk. OS/A+ is compatible with the Atari DOS II and the Basic cartridge.

Basic A+ costs $80.00 and OS/A+ is also $80.00 while the set is $150.00. Either works independently

Mike Dunn, Atari Computer Enthusiasts Newsletter, 3662 Vine Maple Drive, Eugene, Oregon 97405.

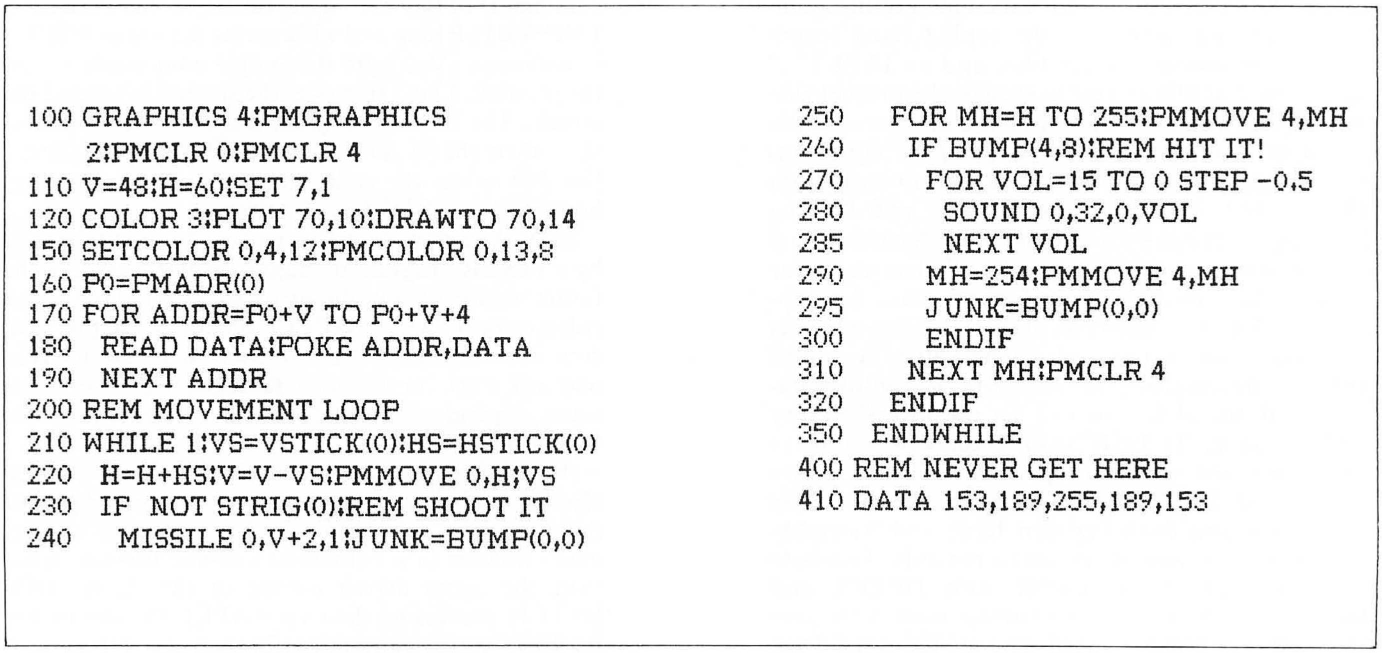
of the other, but the set is an ideal combination. The advantages are many; the disadvantages include the inability to use Atari LOAD (tokenized) files without first LISTing them, and the inability to share your masterpieces with other Atari owners unless they have this Basic. Software developers can get a "runtime" master Basic A+ to include with their programs for a one-time license fee, regardless of the number of copies they sell. Of course, the main competitor to this package will be Atari's Microsoft Basic. The advantages of Microsoft are well known, including the huge amount of published software available, but Microsoft does not have the powerful structured commands available in A+. It does also include a set of player-missile graphics commands. For me, the ease of use of Player-Missile graphics, and the ability to easily interface machine language programs with Basic justify the expense of O.S.S.'s products.

Listing 1.



```
5 GRAPHICS 0
10 SETCOLOR 2,0,0:V=48:H=60
20 PMGRAPHICS 2
30 WIDTH=1
50 PMCLR 0
60 PMCOLOR 0,13,8
70 P0=PMADR(0)
75 FOR ADDR=P0+V TO P0+V+4:
   REM DRAW PLAYER
85  READ DATA:POKE ADDR,DATA
87  NEXT ADDR
90 REM MOTION ROUTINE
100 WHILE 1:REM FOREVER
110   VS=VSTICK(0):HS=HSTICK(0)
120   H=H+HS:V=V-VS:PMMOVE 0,H;VS
130   ENDWHILE
200 DATA 153,189,255,189,153
```

Listing 2.



```
100 GRAPHICS 4:PMGRAPHICS
    2:PMCLR 0:PMCLR 4
110 V=48:H=60:SET 7,1
120 COLOR 3:PLOT 70,10:DRAWTO 70,14
150 SETCOLOR 0,4,12:PMCOLOR 0,13,8
160 P0=PMADR(0)
170 FOR ADDR=P0+V TO P0+V+4
180   READ DATA:POKE ADDR,DATA
190   NEXT ADDR
200 REM MOVEMENT LOOP
210 WHILE 1:VS=VSTICK(0):HS=HSTICK(0)
220   H=H+HS:V=V-VS:PMMOVE 0,H;VS
230   IF  NOT STRIG(0):REM SHOOT IT
240     MISSILE 0,V+2,1:JUNK=BUMP(0,0)
250     FOR MH=H TO 255:PMMOVE 4,MH
260       IF BUMP(4,8):REM HIT IT!
270         FOR VOL=15 TO 0 STEP -0.5
280           SOUND 0,32,0,VOL
285           NEXT VOL
290         MH=254:PMMOVE 4,MH
295         JUNK=BUMP(0,0)
300         ENDIF
310       NEXT MH:PMCLR 4
320     ENDIF
350   ENDWHILE
400 REM NEVER GET HERE
410 DATA 153,189,255,189,153
```

Monkey Wrench
Prehensile Programming

John Anderson

Basic programmers, whether professionals or struggling novices, stand to benefit from any help they can get. Atari Basic is a relatively friendly language in which to work, thanks to extensive syntax checking and a versatile editor. It is an excellent system for learning—yet it has some drawbacks.

Monkey Wrench attempts to correct some of these, and does a very good job of it. It provides nine new Basic commands, as well as a machine language monitor with 15 commands. It also bears the real distinction of being the first (and currently only) ROM board for the right hand slot of the Atari 800.

SOFTWARE PROFILE

Name: Monkey Wrench
Type: Basic Utilities Package
System: Atari 800 8K
Format: ROM "Firmware"
Language: Machine
Summary: Provides several helpful additions to Atari Basic
Price: $49.95
Manufacturer:
Eastern House Software
3239 Linda Dr.
Winston-Salem, NC 27106

Installation

I cannot in good conscience call *Monkey Wrench* a ROM cartridge, as it has no case to speak of. The only disadvantage of this is the possibility of installing it backwards in the computer—a potentially devastating disadvantage. Atari cartridges will not install any way but correctly. Further, the board must be installed with the chips facing *away* from the keyboard—perhaps counterintuitive to the notions of many users. Needless to say, care should be taken on this point.

My machine has been around for nearly two years without ever having anything stuck in the right-hand slot. Hence, when I first plugged in *Monkey Wrench*, I got some rather glitchy results, ranging from a blank yellow screen (you may be familiar with that one, it's an operating system bug), to some spectacular electronic "rain" blowing across the screen.

The manual suggests cleaning the contacts with alcohol. I used a little contact cleaning spray and plugged the board in and out several times. When I looked at the board contacts, they were filthy. I cleaned them with a pencil eraser, plugged the board back in, and got the title display. I then experimented for over an hour without any problems. It is also mentioned in the manual that the 850 interface must be *off* before booting Basic with *Monkey Wrench*.

Operation

Monkey Wrench is "transparent"; that is to say, after the title display indicates that it is functioning, it will not evidence itself again until called. The sole exception to this surfaces when the user tries to move the cursor with "control arrow" keys. The cursor movement arrows are now accessed without the need to press control, while the plus, minus, equal, and asterisk keys are accessed by pressing control.

This option takes a bit of getting used to, but is a much more convenient keyboard configuration for Basic editing. Nine times out of ten, you'll want to use those keys for cursor movement. If this function is for some reason undesirable, you can toggle it off with a simple ">E" command, reverting to normal keyboard operation.

It's Got Your Number

Ready for an editing session in Basic? Well get set for it, because you won't have to worry much about numbering, renumbering, or deleting blocks of line numbers any more. Automatic line numbering is easy; simply enter ">a", followed by your choice of starting line number and the increment value you want. When you press return, those line numbers will be displayed automatically.

This may seem to some to be a minor convenience. All I can say is once you get used to it, you'll never want to be without it.

The same goes for block deletion, accessed by ">d" followed by the first and last line numbers in the range to be deleted. Certainly you could sit and patiently delete each line of the block. In a substantial modification, however, this would become tedious very quickly, and the real benefit of this feature would be seen.

Most powerful and beneficial of the numbering commands is renumber,">R" followed by the starting line value and the increment value you select. *Monkey Wrench* will renumber your Basic program in whatever configuration you wish, changing not only line numbers but all references to line numbers occurring within the program.

I experimented with renumbering three of my own Basic programs (of some complexity), and it works perfectly each time. It should be noted however, that I do not use "names" to call subroutines, a friendly and helpful capacity of Atari Basic, i.e., "GOSUB MAINLOOP." As *Monkey Wrench* is unable to distinguish between "name" constants and any others in a program, this will cause problems in renumbering. This is true of any renumbering routine, and may be reason enough to steer away from "naming," at least when confronted with a choice between quick and painless renumbering and named subroutines.

Because the renumber command uses screen memory as a buffer, there is a limit to the length of a program that can be renumbered. By changing the graphics mode, the total length can be brought to about 1000 lines of code—probably more than you'll need for any single program file.

Some More Than Marginal Additions

By pressing ">M", screen margins can be reconfigured without the need for cryptic POKEs. Since the Atari screen defaults to 38 characters, many programmers (especially those with video monitors) will want to move the margins out to a full 40 characters.

For those who wish to commune with the Atari CPU, the command ">#" will convert decimal values to hexadecimal, while ">$" will convert hex values to decimal. For beginning machine language programmers (of which category I am a lifetime member) these utilities are indispensable.

Typing " >T" followed by hex values will perform a memory test. Don't be shocked if you discover some bad bits of RAM in your Atari. I did, in two machines. The only disadvantage to this function is that testing is very lengthy, and looks just like a system lock-up unless bad bits are turning up.

Monitor Does Not Support Disk

In addition to these commands, a small machine language monitor is provided. Memory location contents can be displayed between any two addresses, and be toggled to display the ATASCII equivalents of these contents, as well as disassembled. The 6502 register contents can be displayed, memory and registers altered, and searches conducted within code for ASCII strings or hex characters.

Memory can be saved and loaded, but very unfortunately, only to cassette. Thus, this monitor will be of only limited utility to all but the most single minded hackers. The monitor is handy for developing short machine language subroutines within Basic programs, and while it

John Anderson is an associate editor for *Creative Computing* magazine.

will run without Basic, it will probably not be of much use in this mode.

The utilities offered by *Monkey Wrench* are easier to use than disk-based utility programs. They never have to be loaded, and are not co-resident with the program you are working on, at least as far as the screen editor is concerned. All commands are available at the touch of a button or two, and with the exception of the RAM test, are uniformly quick to execute. This "transparent" quality will be most appreciated by the intermediate programmer, at whom the package is best aimed.

You will note that I hedged a bit about what *Monkey Wrench* does in and to RAM. The fact is that it does eat up some memory, including part of page six, which could cause some rare problems. Remember also that each cartridge eats up 8K when plugged in: Basic and *Monkey Wrench* will bring free memory on a 48K machine down to about 30K. □

String Arrays in Atari Basic

David E. Carew

Atari Basic differs from most other micro-Basic dialects in its handling of strings. Atari Basic allows strings of any length (limited only by the hardware resource of memory). At the same time an expression like A$(X,Y) in Atari Basic is a substring reference, standing for that piece of A$ beginning at the Xth character position of A$ and running through the Yth position. In many other Basics A$(X,Y) is a string array reference, implying the existence of an array of many strings and referring to the particular string at row X, column Y in the A$ array of many strings.

It is inevitable that those used to reading and programming other Basic's will perceive this difference as a shortcoming of Atari Basic. In fact this is not necessarily a shortcoming at all, but rather a reasonable design decision in implementing a Basic. If the use of substring operations will be more common than the use of string arrays and this is a reasonable assumption for micro-computer applications, then one can eliminate slow and clumsy special function calls such as MID$( ), LEFT$( ), RIGHT$( ) in favor of compact, direct substring references like A$( ). Properly done, this results in a Basic which is faster in executing more common operations. For the occasional application where a string array is needed, it is possible to build your own string arrays in Atari Basic by setting up a single "large" string, and then defining a calculation to convert a row-and-column reference into the correct substring reference for the "piece" of the "large" string corresponding to the row-and-column reference which was made. If you stop and think about it, there are no "rows and columns" in a computer's memory. Those Basic's which provide arrays do so by simulating rows and columns out of a straight list of memory addresses, or positions in memory. We can easily duplicate this behavior by simulating "rows and columns" out of a straight list of character positions in a single, large string. This article is to show exactly how this can be done.

Suppose we wish to have a string array 4 rows by 3 columns, with each string in the array having a maximum length of 20 characters. We start by setting these quantities up in variables:

```
100 ROWMX = 4:  COLMX = 3:
  LNGMX = 20
```

Given these quantities, we know how long to make our "array" string:

```
150 TTSIZ = ROWMX * COLMX
  * LNGMX
200 DIM ARR$(TTSIZ)
```

We could perform the reference conversion calculations each time a reference is made in the program, but since each repeat of a particular reference would imply a repeat of exactly the same calculation, it is more efficient as well as more convenient to perform the conversion calculations once and store the results in such a way that they are easily accessed as needed. One table (numeric array) for the beginning substring positons and one for ending substring positions allows for convenient addressing; and this is illustrated below:

```
206 REM BG IS BEGIN SUBSTR
  TABLE, EN IS END SUBSTR
210 DIM BG(ROWMX,COLMX)
220 DIM EN(ROWMX, COLMX)
230 REM INITIALIZE "STR$
  ARRAY" CONTROL TABLES
240 FOR RW=1 TO RDWMX :
  FOR CL=1 TO COLMX
250 BG(RW,CL)=COLMX *
  LNGMX * (RW-1)+(LNGMX*
  (CL-1)+1)
260 EN(RW,CL)= BG(RW,CL)-1
  + LNGMX
270 NEXT CL: NEXT RW
```

The only step remaining would be to initialize ARR$ to all blanks (or some other appropriate filler).

Having made these extra arrangements to start with, then every occurrance of another Basic's ARR$(X,Y) expression might be replaced with an Atari Basic equivalent:

```
ARA$(BG(X,Y),EN(X,Y))
```

This solves the address conversion part of the problem. A detail or two may remain. In most string-array Basic dialects, ARR$(3,4) may have a length of zero, or any other length up to some maximum. In Atari Basic, using string-array simulation, ARR$(3,4),EN(3,4)) has a length of LNGMX exactly, no more and no less. The consequences of this detail depend on the application. For instance, a string-array Basic may test for an empty array cell using a LEN function, like this:

```
6000 If LEN(A$(3,4))= 0 THEN . . .
```

The equivalent array-simulation code might involve a string of length LNGMX initialized to all blanks. Then an empty cell is not LEN equal zero, but rather equal to the "always empty" string, e.g.:

```
6000 IF A$(BG(E,Y),EN(X,Y)) =
  NUL$ THEN . . .
```

Also, placing a string shorter than LNGMX into a simulated array may require taking its length into account.

```
7000 ARR$(GB(X,Y),BG(X,Y)
  -1+LEN(NEW$))=NEW$
```

The above code places a short (i.e., LEN(NEW$) =LNGMX) NEW$ into the X,Y cell of ARR$, beginning at the

David E. Carew, Interactive Management Systems Corp., 3700 Galley Rd., Colorado Springs, CO 80909.

first character position of the cell and taking as many positions in the cell as required by the length of NEW$. This statement is obviously longer, less intuitively clear and certainly somewhat slower executing than the non-Atari Basic equivalent:

```
7000 ARR$(X,Y) = NEW$
```

However, the simulation still provides a single statement, directly substitutable for the non-Atari equivalent, if for example you are covering a listing from some other Basic. I have found that other details I have encountered are similarly susceptible to fairly happy solutions.

The next time you have an application which cries out for string arrays (or a possible conversion of a listing which already uses string arrays) you might consider the approach suggested here. Once you have mastered string array simulations for the relatively rare situations where you actually need them, then Atari Basic's compensating payoff of quicker, cleaner substring manipulation seems all the sweeter.

Talk is Getting Cheaper

John Anderson

Giving your computer the power of speech is no mere frill or gimmick. The potential of such capability, for the handicapped as well as microcomputer users at large, is dramatic.

For as long as microcomputers have been around, the cost of such potential has remained a prohibitive factor. But that is changing fast.

Following is a look at three speech synthesis packages for the Atari computer. These packages represent the range of possible configurations: the first is an independently powered piece of hardware, which can hook up to any microcomputer using a serial or parallel port; the second consists of an Atari specific external module, driven by software; the third works entirely in software, using the synthesizer chip already in the Atari.

The Echo GP

I have had an opportunity to experiment with the Echo Speech Synthesizer, from the Street Electronics Corporation, for quite a while now. It is a sophisticated unit, while at the same time fun to use.

It is based on the Texas Instruments TMS 5200 speech processor chip. This is in contrast with its nearest competitor, the Votrax Type 'n Talk, which uses the Votrax chip.

The unit makes use of its own 6502 microprocessor, and interfaces as if it were a printer. It is available in RS-232 serial or Centronics parallel versions. This means that the 850 interface is needed to drive the Echo from an Atari computer. We received the serial version, and controlled it through the 850 using Atari Basic.

Upon power-up, the Echo unit responds with the phrase "Echo ready," to let you know all is well. One of the first points the user will notice is that the Echo is capable of intoning a sentence. Rather than speaking in monotone, the pitch of the voice is dynamic. This makes for a more intelligible and less grating speech quality.

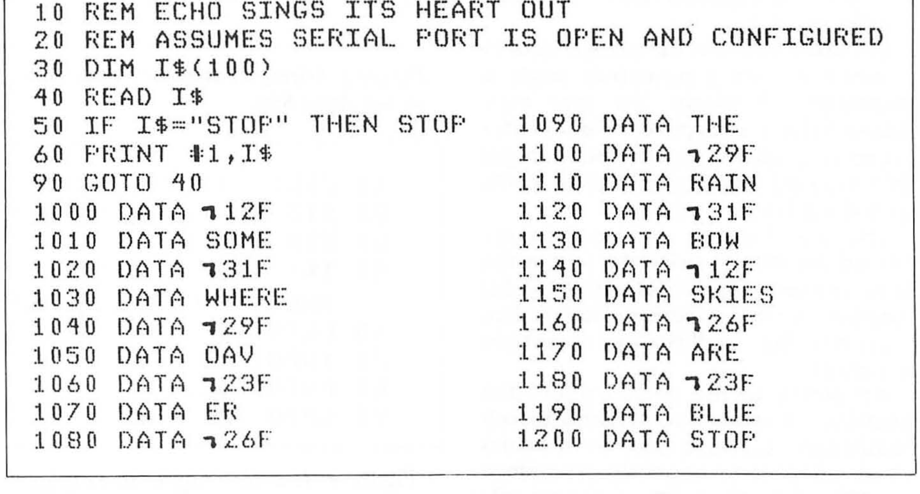
You can use the internal speaker of the unit or route the sound to an external speaker. I found it convenient (as did those around me) to use an earphone when involved in speech editing sessions.

Textalker

Textalker is the ROM based program Echo uses to convert English into speech. Echo can translate English text into phonemes directly, with an impressively low error rate. It can be disorienting, but even when Echo mispronounces a word or syllable, the listener can usually make sense of the sentence from its context.

		Do	Re	Mi	Fa	So	La	Ti	Do
Octave 1	-	12	15	18	20	23	26	29	31
Octave 2	-	31	34	37	39	44	48	51	53
Octave 3	-	53	56	58	61	63			

Figure 1. A rough pitch table to give the synthesizer a singing voice. Flats and sharps can also be supported, but I have not taken the time to locate them.



```
10 REM ECHO SINGS ITS HEART OUT
20 REM ASSUMES SERIAL PORT IS OPEN AND CONFIGURED
30 DIM I$(100)
40 READ I$
50 IF I$="STOP" THEN STOP       1090 DATA THE
60 PRINT #1,I$                  1100 DATA ┐29F
90 GOTO 40                      1110 DATA RAIN
1000 DATA ┐12F                  1120 DATA ┐31F
1010 DATA SOME                  1130 DATA BOW
1020 DATA ┐31F                  1140 DATA ┐12F
1030 DATA WHERE                 1150 DATA SKIES
1040 DATA ┐29F                  1160 DATA ┐26F
1050 DATA OAV                   1170 DATA ARE
1060 DATA ┐23F                  1180 DATA ┐23F
1070 DATA ER                    1190 DATA BLUE
1080 DATA ┐26F                  1200 DATA STOP
```

Figure 2. With a singing synthesizer your micro won't be in Kansas anymore. The character " ┐ " is what control-e looks like on the screen.

John Anderson is the associate editor of Creative Computing magazine.

This is not to say that Echo has the diction of Henry Higgins. In fact, it takes a bit of time to become accustomed to the unique "accent" of the unit. As is the case with some foreign speakers, accustomed listeners will typically understand words that first-time listeners will miss. Echo has trouble with the "g" sound in words like "go," and "l" sounds give it problems as well.

In this respect, the monotone of the Type 'n Talk wins out. (A thorough review of the Votrax unit appears in the September 1981 issue of *Creative Computing.*) Though it also has its share of vocal peculiarities, it does, on the whole, enunciate more clearly than the Echo. And yet, for extended periods, I would much rather listen to the Echo. The monotone of the Votrax unit gets me down after a while—too "computerish." It was an unfortunate design decision. The Votrax chip itself, as we shall soon see, does allow for software pitch control which results in much more natural sounding speech.

The features of Echo are accessed through control characters. For instance, pressing CONTROL-E will enable the Textalker command set. Following this character with a number from 1 to 63 will determine pitch, which can be toggled from f (for flat, meaning unintoned), to p (for pitched, meaning intoned). In what I think is a first for microcomputers, I found that the Echo could be programmed to "sing" through careful use of these commands. In fact, the unit provides for about three octaves. Not a bad range! A pitch table and sample program appear below.

be controlled by text punctuation. A comma will create a pause, a period will cause a drop in pitch at the end of a sentence, and a question mark will result in a rise in pitch.

Textalker can also be commanded to pronounce each punctuation mark it encounters. Similarly, the user may choose to have all upper case letters pronounced as letters (use this mode to get IBM to sound right), or to have all words spelled out letter by letter.

The rate of speech may also be compressed resulting in twice the text in the same amount of time. Remarkably, this function sometimes increases rather than decreases the intelligibility of certain sentences.

According to the documention, the Textalker component of the Echo Speech Synthesizer "contains close to 400 rules which allow it to correctly pronounce over 96% of the thousand most commonly used words in English."

I was pleasantly surprised at how well Echo did with unaltered text. Having worked with phonemically-based sythesizers in college, I realized this was quite a feat. Of course there are some words Echo has trouble with. Fortunately, an appendix, which outlines the kinds of fixes to apply to these words, is provided. They are as simple as the addition of a space, such as "cre ate" for the word "create," or the spelling of the word "question" as "kwestchun."

Phoneme Generator

In addition to the Textalker module, speech can be programmed at the phonemic level, using the Speakeasy Phoneme Generator, also resident in firmware. This mode is selectable by the character CONTROL-V, and provides for much more detailed control. Stress, pause, pitch, volume, and rate controls can be embedded directly into the text strings.

This approach requires the use of a phoneme code, detailed in the documentation. It bears little resemblance to any phonetic alphabet I have come into contact with, but the 48 sounds it provides are more than enough to do the job.

Male DB-9 - (to serial - port #1)	Male DB-25 (to Echo GP)
Pin 1	No connection
Pin 2	No connection
Pin 3	Connects to pin 3
Pin 4	Connects to pin 2
Pin 5	Connects to pin 7
Pin 6	Connects to pin 20
Pin 7	Connects to pin 5
Pin 8	Connects to pin 4
Pin 9	No connection

Figure 3. Wiring a cable for connection to the Atari 850.

Unfortunately, the effort it takes to achieve satisfactory results using this approach is somewhat unreasonable, especially in contrast to the serviceable job Textalker does. However budding linguists should take note. The phonemic approach offers great experimentation potential. I did manage to get the Echo speaking a little German.

The Echo Speech Synthesizer lists for $300, which is admittedly a bit stiff. Still, it is comparable to the price of the Type 'n Talk. And if you want your micro to sing Thomas Dolby tunes, the Echo is the only choice.

Hooking Up

In March of this year *Creative* ran a review of the Echo Speech Synthesizer board for the Apple II. At that time, Textalker and Speakeasy were in the development stage. The Speech Synthesizer offers much greater flexibility and power, as well as the capability for connection to any personal computer.

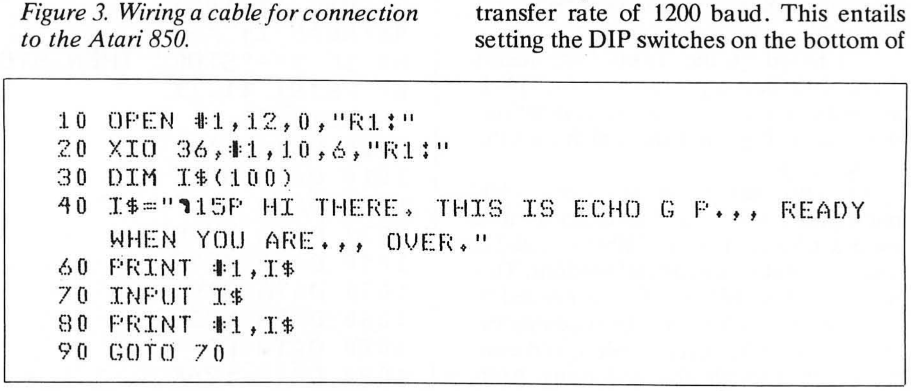
However this does not automatically imply *easy* connection. Even with our experienced people here at the magazine, it took us a while to make the Echo conversant with the Atari.

The documentation that arrived with our Echo was preliminary. All the information we needed was there; I do hope that the final documentation will be an improvement, though.

The real fault lies with the 850 interface module documentation: it provides beginners with quite a run for their money. Here is a way to succeed.

The first thing to do is wire an interface cable, by connecting a DB-9 male to DB-25 male connector. The pinouts given in Figure 3 work with serial port number 1 on the 850.

Next you need to configure the Echo and port number one so that communication may be established. I used a data transfer rate of 1200 baud. This entails setting the DIP switches on the bottom of



```
10 OPEN #1,12,0,"R1:"
20 XIO 36,#1,10,6,"R1:"
30 DIM I$(100)
40 I$="˥15P HI THERE. THIS IS ECHO G P.,, READY
   WHEN YOU ARE.,, OVER."
60 PRINT #1,I$
70 INPUT I$
80 PRINT #1,I$
90 GOTO 70
```

Figure 4. It is this simple to configure serial port number one and input text for synthesis. Again the " 7 " character signifies control-e. Don't forget to boot the device handler prior to running the program.

the Echo so that positions 1 and 2 are on, while position 3 remains off. Position 4 also remains in the off position to enable "handshaking," as we say in the trade.

The serial port is configured through software. Figure 4 shows an example of this configuration, as well as a short program allowing for straightforward experimentation with the unit.

Make sure the 850 device handler is booted whenever using the serial port. This occurs as an autorun.sys file on the Atari DOS disk. Make sure it is resident on any program disk for use with the unit. Power up the 850, then boot a disk with the handler file. You will then be set to go.

For more information concerning the Echo, contact Street Electronics, 1140 Mark Ave., Carpinteria, CA 93013.

The Alien Group Voice Box

The Echo has everything it needs to effect speech synthesis onboard. Like a printer, it awaits a stream of characters; it would just as soon pronounce text files from bulletin board services, Compuserve, or the Source. The Atari, thus, is free to do whatever processing you have in mind, while the Echo works independently.

This is a fine capability, but also an added expense. The Voice Box from Alien Group takes some of the internal, ROM based capabilities of the Echo, and efficiently uses Atari RAM for their storage. The Voice Box uses a Votrax SC-01 chip, and connects directly to the Atari input/output jacks. It will necessarily be the final connection in the I/O daisy chain, as it offers no jack of its own.

The external module is no bigger than a transistor radio, and draws power directly from the Atari. It lists for $170, including driver software, which is available in cassette or disk versions.

The Voice Box is manipulated from Atari Basic, and does not offer an RS-232 handler program. Using patches from Basic, however, it can be controlled from a machine language program.

Your machine must have at least 16K to run the Voice Box. If you have 32K or more, you can run two additional programs included with the package: the Random Sentence Generator and the Talking Face. More about these later.

When the driver program is run, the box responds with the phrase "Please teach me to speak," or if a dictionary is loaded, the words "Yes, Mahster," to let you know everything is working.

While calling on its own phonetic input code, as does the Echo, the system also uses a unique approach to convert character strings into speech sounds. English text and phonetic code may be freely intermixed, rather than requiring separate modes, as is without exception the case with every other text-to-speech system I have seen.

Dictionaries

The key to working with the Voice Box is the creation of your own *dictionaries*. These are the "word equations" specified to translate words into phonemes. For example, by typing "spek=speak," you will ensure that each time the word "speak" is encountered, it will be pronounced correctly. Dictionaries are saved and re-called, as independent files, to cassette or disk. In addition to those you create, three pre-written dictionaries are supplied with the driver software.

Dictionaries eat up computer memory quite quickly—each word equation takes up ten bytes. In order to store phonemes more effeciently, word *fragments* can be stored. You can define fragments to be recognized only at the beginning or the end of a word, or at every occurrence.

Because dictionary size is limited, the dictionary approach itself is necessarily limited. Even with 48K, no dictionary is going to produce impressively accurate text-to-speech capability. In this respect, the Echo has a much more sophisticated algorithm. This is the main trade-off between the two systems.

In fact, if you have more than 32K, you must change the dimensions of a string statement in the Voice Box driver program in order to store larger dictionaries. The documentation clearly states how to do this.

Other Features

Similar to the Type 'n Talk, the Voice Box sports a potentiometer knob on the front of the case, that can be used to vary the speed and pitch of the speech. The Voice Box unit allows for pitch control through software, too. Control is restricted to four registers, utilizing the

SAM Speaks Apple II

The Apple II has no special advantage over the Atari when it comes to speech synthesis. The Echo, Votrax, and many other voice systems work equally well for both computers.

The history of software-only synthesizers for the Apple dates back to 1979 when Softape published a program called Apple Talker. That program has been discontinued, but Muse publishes The Voice, an inexpensive program that serves the same purpose. Sirius Software, the renowned game publisher, produces Audex, a general purpose audio program that can be used to approximate speech. For the most part, these programs deliver results that are interesting, but only sporadically intelligible.

Hardware voice products for the Apple also abound. Voice input can be recognized by peripherals from Scott Instruments, among others. Mountain Computer carries a remarkable input-ouput device that turns an Apple into a digital audio recorder.

At $130 for the Apple version, SAM is the first product to combine unlimited vocabulary, impeccable intelligibility, and reasonable price.

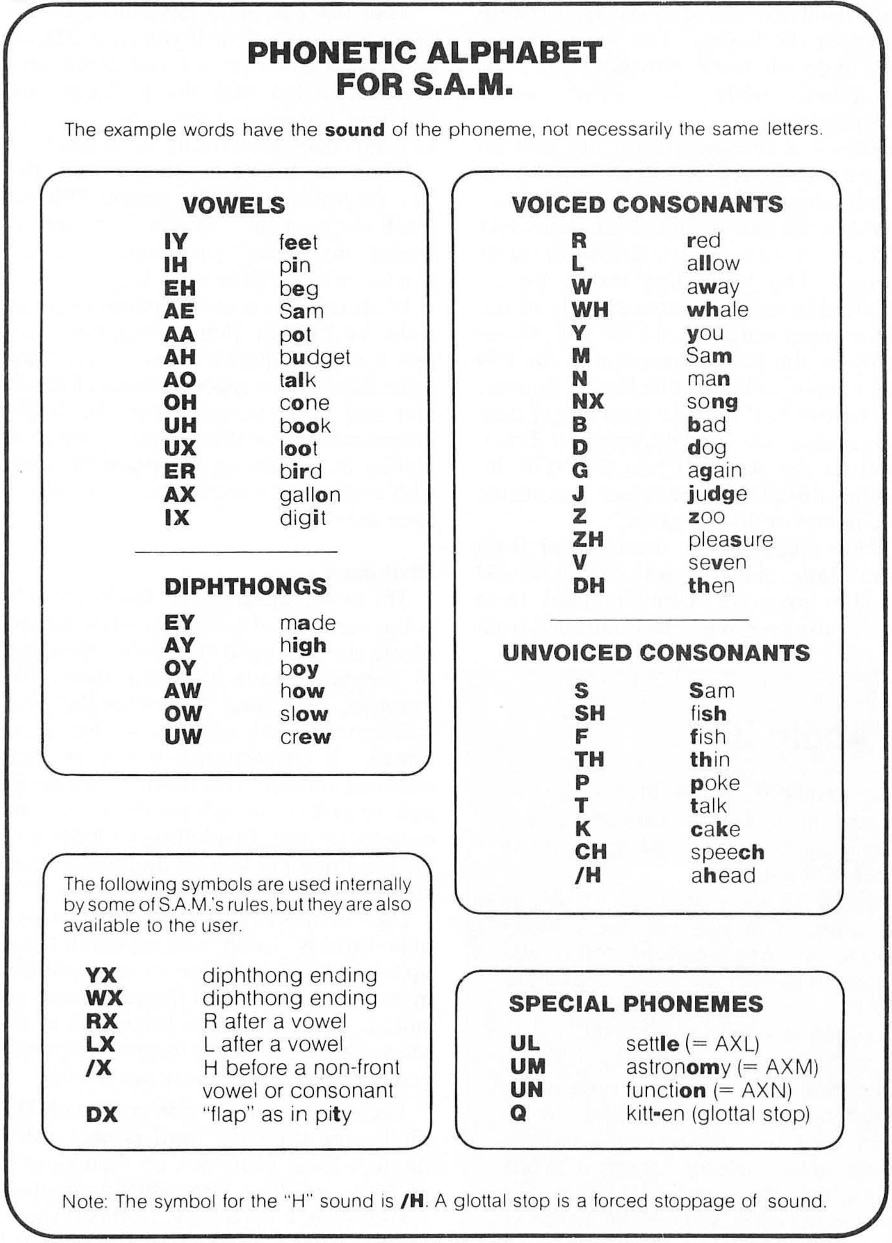
The SAM package includes a little bit of hardware and a little bit of software. The hardware is a board containing a digital-to-analog converter, a tiny amplifier, and an even tinier volume control. The software includes all the programs described in the main part of this article.

SAM sends output to an 8-ohm speaker. You can use the speaker inside your Apple or, for better results, attach a slightly larger one. Installing SAM is no more complicated than hooking an Apple to a TV set.

SAM uses the simplest possible interface to a sound system. In exchange for the simplicity of the hardware, the developers had to write large and complex programs. The program that produces speech based on phonetic codes occupies 9K of RAM. Another program that translates English text into phonetic codes requires an additional 6K. These programs live in an area usually reserved for Applesoft string variables. The English translator also overlaps the memory associated with the second graphics image (Hi-Res page 2) of the Apple.

Because of these requirements, SAM can not cooperate with most other programs. You can not add speech capability to your word processor or terminal program, for example. Pascal, Logo, Graforth, and most other languages can not use SAM.—*MC*

Figure 5.



PHONETIC ALPHABET FOR S.A.M.

The example words have the **sound** of the phoneme, not necessarily the same letters.

VOWELS

IY	f**ee**t
IH	p**i**n
EH	b**e**g
AE	S**a**m
AA	p**o**t
AH	b**u**dget
AO	t**al**k
OH	c**o**ne
UH	b**oo**k
UX	l**oo**t
ER	b**ir**d
AX	gall**o**n
IX	dig**i**t

DIPHTHONGS

EY	m**a**de
AY	h**igh**
OY	b**oy**
AW	h**ow**
OW	sl**ow**
UW	cr**ew**

VOICED CONSONANTS

R	**r**ed
L	a**ll**ow
W	a**w**ay
WH	**wh**ale
Y	**y**ou
M	Sa**m**
N	ma**n**
NX	so**ng**
B	**b**ad
D	**d**og
G	a**g**ain
J	**j**u**dg**e
Z	**z**oo
ZH	plea**s**ure
V	se**v**en
DH	**th**en

UNVOICED CONSONANTS

S	**S**am
SH	fi**sh**
F	**f**ish
TH	**th**in
P	**p**oke
T	**t**alk
K	**c**a**k**e
CH	spee**ch**
/H	a**h**ead

The following symbols are used internally by some of S.A.M.'s rules, but they are also available to the user.

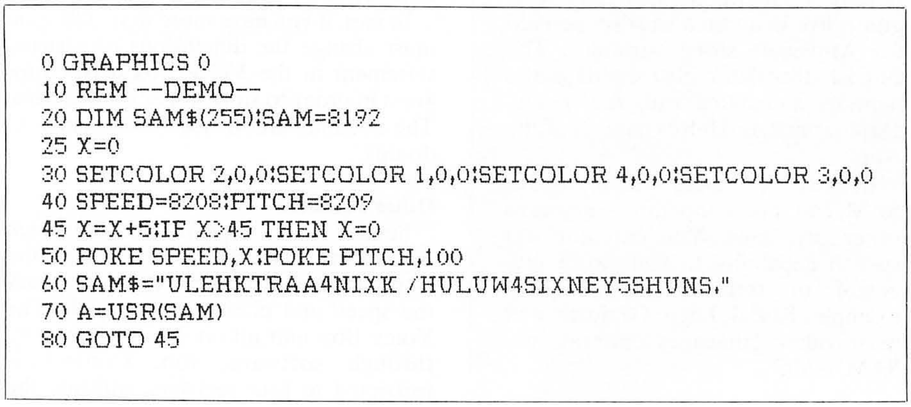
YX	diphthong ending
WX	diphthong ending
RX	R after a vowel
LX	L after a vowel
/X	H before a non-front vowel or consonant
DX	"flap" as in pity

SPECIAL PHONEMES

UL	sett**le** (= AXL)
UM	astrono**my** (= AXM)
UN	functi**on** (= AXN)
Q	kitt-en (glottal stop)

Note: The symbol for the "H" sound is **/H**. A glottal stop is a forced stoppage of sound.

Figure 6.



```
0 GRAPHICS 0
10 REM --DEMO--
20 DIM SAM$(255):SAM=8192
25 X=0
30 SETCOLOR 2,0,0:SETCOLOR 1,0,0:SETCOLOR 4,0,0:SETCOLOR 3,0,0
40 SPEED=8208:PITCH=8209
45 X=X+5:IF X>45 THEN X=0
50 POKE SPEED,X:POKE PITCH,100
60 SAM$="ULEHKTRAA4NIXK /HULUW4SIXNEY5SHUNS."
70 A=USR(SAM)
80 GOTO 45
```

slash and the backslash characters to move between them. This negates the musical capabilities of the unit, but is a step ahead of the monotone of the Type 'n Talk.

Because so much of the Voice Box is RAM resident, you must decide how much of the memory of the Atari to allot to dictionary space, in addition to your own Basic programs, and the Voice Box driver. The disk version includes a pared-down driver program for incorporation into other programs. The documentation also gives hints for memory conservation.

In the 32K version, several other features appear. The first is the Random Sentence Generator. The Voice Box will compose random but grammatically correct sentences from its stored word lists. These can be modified with word lists of your own creation. I obtained some rather strange results in my attempts at this. While many were semantically bizarre, I must. admit the sentences were grammatically unassailable. Be prepared for a few shocks when you try this.

There is also a mode called The Talking Face. This displays an animated face, with impressive lip synch simulated as words are articulated by the Box. I am sure this feature would be a big hit with the kids.

The documentation accompanying the system is a bit uneven in places, but manages to cover all the features of the Voice Box in a scant nine pages. The phoneme list is quite complete. The documentation also goes as far as to suggest to assembly language programmers a means of updating data to the box while running machine language animation routines.

While the Voice Box is not really in the same league as the Echo, it offers many of the same features for much less money. For more information contact the Alien Group, 27 West 23rd St., New York, NY 10010.

The Software Automatic Mouth

In the September 1982 issue of *Creative*, I mentioned that the Atari was capable of speech synthesis using only its internal hardware. The game *Tumblebugs* taught the Atari its first words: "We gotcha!" This came as a happy revelation to many.

Well with *Software Automatic Mouth, SAM* for short, Mark Barton has brought this possibility to fruition. He has created a disk-based, unlimited speech synthesis program, requiring *no* external hardware. And the speech quality of *SAM* competes favorably with the *best* systems available for microcomputers.

SAM uses the Atari sound chip, Pokey, to generate speech. Even with my unbridled faith in the capabilities of the Atari, I was quite surprised at how well it does the job. Pokey is at least as intelligible as its two competitors, the TI and Votrax chips.

SAM is the only package around that dares to include lengthy prepared speech demonstration programs to show off its articulative powers. My colleagues agreed that no break-in period was necessary in order to understand *SAM*.

The documentation supplied is equally impressive. It not only makes operation of the program very simple, but provides background information concerning linguistics and speech synthesis. It helps to make the program into an excellent tutorial on the subject.

I did encounter one snag, if only in my eagerness to get rolling with the package. You must copy all the Basic programs from the master disk to a new diskette. The autoboot assembly language program that constitutes *SAM* runs from the master, but support programs must be loaded from the new disk. The reason is that the support programs require a mem.sav file. The write-protected master disk will, of course, return an error if a mem.sav attempts to write to it. The documentation clearly states that you must use an un-write-protected new disk with a mem.sav file on it. In my excitement to get going, I did not heed these instructions, and ended up wasting some time.

Support programs included with the package are: Reciter, which is an English text-to-speech translation program; Sayit, the short Basic program which makes experimentation simple; Demo and speeches, two files that impressively demonstrate the powers of *SAM*; and Guessnum, a spoken version of a number-guessing game.

An RS-232 handler program is also provided, allowing *SAM* to act as Echo does to read telecommunications text.

It is extremely simple to work with *SAM* from Basic. All that is needed is to define SAM$ as it appears in Basic, and then invoke either *SAM* or Reciter through a USR call. You can also effect machine language patches from Basic.

Speech Quality

The really remarkable thing about *SAM* is its (his?) intonation—*SAM* can be extremely expressive. Control of stress placement is easy. The phonetic code is a bit strange, but very nicely laid out in the documentation (see Figure 5). A reference card is also provided.

Similar to the Echo, punctuation is "understood." A hyphen is read as a short pause, and is handy for delineating clause boundaries. A comma inserts a pause equivalent to two hyphens. A question mark also inserts a pause, as well as making the pitch rise at the end of a sentence. Likewise a period makes the pitch fall.

SAM is capable of speaking only 2.5 seconds without a break. If a string exceeds that length, a short break will automatically be inserted. If you don't like the placement of automatic breaks, you can stipulate their positions with hyphens. The breaks are so short as to be hardly noticeable, and cause few problems.

SAM can be controlled more creatively and flexibly than Echo or Voice Box. The pitch and speed of *SAM* speech can be altered through with POKE statements. I got some wild results playing with these. A sample program, Figure 6, shows how speed effects can be achieved.

The timbre of speech can be varied to make *SAM* sound quite human—or like a droid from *Star Wars*.

An 18-page English-to-phonetic code dictionary appears in the documentation to help in speech programming. In addition, *SAM* flags phoneme input errors. When a bad phoneme occurs in the immediate execution mode, an error is flagged in the same way as syntax errors in Basic. By PEEKing decimal address 8211, you can trace these problems when they occur in the deferred mode.

At the incredible price of $60, there must be a catch, right? Well there is, sort of. Because *SAM* uses the Atari to do all its work, DMA is shut down during articulation. This means the screen goes blank ˬuring speech—no animation, no text, nothing. The documentation tells you how to re-enable DMA during speech, but warns that this distorts *SAM's* speech rather badly. However, this blanking takes place only during articulation. As soon as a string is finished, DMA returns and all is normal.

I cannot overstate how impressed I am with the *Software Automatic Mouth*. It is a remarkable feat of software savvy, and probably one of the best buys available for the Atari computer. Its higher-priced competitors have their advantages, but would do well to strive for the same strong documentation this package has. If you wish to give your Atari the power of speech, have a disk drive, and are on a limited budget, look at this program. For more information, contact Don't Ask Software, 2265 Westwood Blvd. Suite B-150, Los Angeles, CA 90064. □

Axlon RAMDisk
128K Memory System for Atari

A while ago a group of employees left Atari and formed Axlon Co. to manufacture add-on products for the Atari computer. They produce a 32K RAMCram card, and a 256K RAM system, complete with expansion interface.

So when ads began to appear for their RAMDisk, I was intrigued. I couldn't resist calling them for more details.

They turned out to be most friendly and mailed me a loaner RAMDisk for evaluation. This review is based on my use of the product for a month.

The RAMDisk arrives in a 9" x 11" x 1" blue box which contains a manual, a diskette, and a memory cartridge. The manual is housed in an attractive notebook with the diskette in a side pocket.

The memory board looks like an Atari 16K cartridge except that it has no top or sides. (It does have front and back covers, though.) This is probably to help the RAMDisk get rid of heat.

The manual is well written, and very, very clear. I decided to trust it immediately, and began following the setup directions.

Getting Started

First, one boots up the system with a normal Atari 2.0S DOS disk. The Axlon disk is fast formatted and uses Atari directory formats and such, but does not contain DOS.SYS or DUP.SYS, so you can't boot up with it. Next, I ran a Basic program called CREATE to create a boot disk (a disk used whenever the system is powered up or re-booted). Following the instructions, I put in the Axlon disk, then a blank disk, and created a boot disk. No problem—very easy to do.

Next, I turned the computer off, removed my middle 16K board, and put in the RAMDisk. Two memory boards are required, for some reason, on either side of the RAMDisk. Perhaps they keep it from getting the electrical equivalent of lonely.

Then, I booted up using the new boot disk. A most foreboding message flashed onscreen for about five seconds, just long enough for a speed reader to comprehend it. It pointed out that the Axlon MMS (Memory Management System) was an end-user initiated change to Atari DOS and that Axlon doesn't condone making copies and distributing them.

Here is my first complaint with the MMS, Axlon's DOS 2.0S: you have to sit through this silly legal message *every* time you boot up with the MMS disk. The first time, it's fun, and even witty. The second time, half witty, and after that, not funny at all. I was ready to disassemble the boot file and "short out" the message after a month of seeing it.

Once you're through the message, you get to Basic or whatever you're running. The RAMDisk hardware operates just like a normal 16K cartridge unless you specifically tell it not to. There's just about the same amount of free memory as before. So I typed DOS.

Next surprise. No click, whirr of the disk. The DOS menu popped up right away, just like the old DOS 1, but apparently without the memory sacrifice, according to FRE(0). And my, how the DOS menu has changed.

The Menu

First, the top line is not Atari 2.0S anymore. It is the Axlon RAMDisk MMS System V1.0. Most of the options look the same, but two are disabled: writing DOS files and creating MEM.SAV.

Second complaint. I don't care about MEM.SAV; I never use it. But I want to be able to write the DOS files after formatting a disk. The DOS and DUP files are on nearly every disk I have, making for few bootup problems. But Axlon doesn't want complaints about folks copying DOS, so they disabled it. Aside from these two changes the menu is a duplicate of the Atari 2.0S menu.

How do you use the RAMDisk? The RAMDisk contains 128K bytes of memory. A diskette contains around 90K. So the Axlon MMS makes "disk #4" the RAMDisk memory area. You literally use the 90K of the memory board as disk number 4.

You can copy to it, open files on it, close them, NOTE/POINT them, and so forth. You can copy an entire disk to RAM. You can run directories, lock files—everything you can do to a normal disk—to the RAMDisk (disk 4). In short, the RAMDisk replaces disk 4.

Here's an example. Let's say I have one disk drive and I need to duplicate a disk. I load the Axlon MMS, go to DOS, and J (duplicate disk) from 1 to 4. This copies the whole diskette into RAM. Next, I put in my destination disk, and copy from 4 to 1. All done. (No more swapping diskettes back and forth.) This is very nice and very easy. It is also fast. I could load 220-sector binary files in less than a second from the RAMDisk. This compares to more than 30 seconds for a disk drive.

Software houses should take note here. The RAMDisk is a very good thing for you. Let's say you need to make 100 copies of a given diskette. Without the RAMDisk, you can either use two drives—one to read the master and one to write the destination disk (wearing the master and its drive out)—or use one drive and swap disks like mad. With the RAMDisk, you copy the master into RAM, then proceed to make your copies from RAM. This product would well pay for itself in saved time and disk drive wear—heavy use is hard on Atari drives. (By the way, I found that DOS and DUP did copy if I used the DUP DISK option; you just can't create them originally).

From Basic Assembler, and so forth, the RAMDisk is just disk 4. SAVE or LOAD; the operations run very fast. Anyone with a program that is running slowly due to disk I/O should look into the RAMDisk. A speedup factor of 20 would be easily achieved, and that's conservative. In addition, you needn't put up with disk errors and the like.

How It Works

By now you're probably curious how this thing works, so here's what I found (in the manual, all clearly laid out). In the Atari, the address space from 4000 to 7FFF is normally the second 16K board installed in the machine. The RAMDisk allows 4000-7FFF to be any of eight individual 16K boards, one at a time. Due to many arcane hardware considerations you can't access all 128K at once, only a 16K chunk of it. But *which* 16K is instantly selectable. This is called "bank selection."

For example, Axlon apparently puts their MMS DOS Menu on one of the 16K banks. Then, to switch to DOS, they just select that particular 16K, and run (that's why DOS comes up so fast). But also note that DOS does not take up normal 16K programming space this way; the contents of the 16K you were working in before you typed DOS are on another of the 16K boards, ready for use as soon as it is reselected. (The MMS handles the swapping back and forth to use the 90K disk area).

If you're confused, just imagine you have a pile of eight 16K memory boards and you could plug or unplug them at will into the middle slot. This is how the Axlon board works.

Physically it uses Motorola 64K x 1 chips. The raw cost of the chips on the board I calculated to be around $250, so the price of the board is quite reasonable. The construction of the board is very high quality.

Uses

Extremely high speed animation is possible using bank selection. You don't have to use the Axlon board as a RAMDisk. You can select which 16K you want directly. So several images (display lists and memories) can be stored, and switching between them determines which image is being displayed. Some impressive effects could be obtained (only) this way. Alas, I didn't have time to do much of this.

One thing I did use the system for was holding temporary files during developmental work. By having the RAMDisk hold various versions of a Basic program I was developing (with SAVE), I greatly speeded up the development time. However, there is a problem with this: turning the Atari off causes the contents of the board to be lost. And I have locked up the Atari past RESET working many, many times.

The diskette that comes with the RAMDisk also has several options to check the board out and fiddle with MEM.SAV. It even has a complete copy of the manual (over 300 sectors) as files.

Disadvantages

And now I come to the parts I don't like about the RAMDisk.

I have already mentioned a few points, but my main problem with this unit is that it is a limited function device. It is like a plotter; some people can use it, others can't. Software houses and people with heavily disk-bound programs could make great use of this product. People who need incredible animation memory also could. But I can't for the life of me think of another use for it. It was a nice convenience when copying disks, but it just wasn't that great a help. It would take a volume operation for it to make a difference. For your average Atari user, another disk unit, which costs the same (or even a bit less) is probably a better buy. You can just do more with it.

Technical Aides

The bank selection is done in the C000 area, currently unused by Atari. My Atari sources tell me this will change in a year or so, as the operating system acquires more capabilities. The Axlon people will have to modify their board at that time.

Sector copying programs do not work with this board.

Microsoft Basic has real problems with this board. I tried the whole month to get them to work together and couldn't. As the new Basic is just plain wonderful and everyone will be buying it, the Axlon people had better get some new software out fast.

The board throws only minimal RF interference, and if you run your Atari without the top cover on for heat dissipation, you will notice minor wavy lines on your TV.

Axlon plans a RAMDisk for Apple II and Apple II Plus computers in the near future.

Conclusion

This is a solidly built, well documented product. It has several very useful applications. People who can use it in those applications will be most pleased with it. But those who want high speed disk I/O or temporary storage will not find it of much use. It certainly expands the capabilities of the Atari, but you may not need your capabilities expanded in that direction. Consider it as you would a piece of other special purpose peripheral equipment, such as a digitizer or modem. Will you use it? If so, it is a good product.

RAMDisk, Axlon Co., 170 N. Wolfe Rd., Sunnyvale, CA 94086. $699. □

Joytricks

John Anderson

Ever stare at the controller jacks in the front of your Atari computer and imagine all sorts of exotic hardware to connect up to it? I have, and while my work on a fully articulated robot arm is progressing quite slowly, there are a few modification projects I've undertaken that require little time, cost very few dollars, and provide nice results.

End Discrimination Against Lefties

As a left-handed gamesman, I've long suspected that my scores have been held down by the fact that joysticks are designed for righties. It's a very simple matter to turn a standard issue Atari joystick (fire button top left) into a lefty stick (fire button top right).

When you disassemble the joystick, be careful not to lose any of the screws or the little spring that sits in the trigger button. Hold the circuit board so it resembles the configuration in Figure 1. Note: newer Atari joysticks have all the connectors on one side of the PC board while older ones have three connectors on each side.

The leads must be removed from the board (grasp the collars; do not pull on the wires themselves) and reattached as shown. That's all there is to it—except to prominently label your new lefty joystick so that it does not drive some poor righty mad. The stick is now "referenced" with the trigger to the upper right.

A Pushbutton Peripheral For Under $8

I've been thinking about a homebrew controller jack peripheral for quite some time now, but the genesis of this idea really belongs to Rick Rowland. Though the controller is at its best when playing a limited number of games, you can do quite a bit with it. If you have a joystick that has seen better days and is ready for retirement, you can reincarnate it as a pushbutton peripheral.

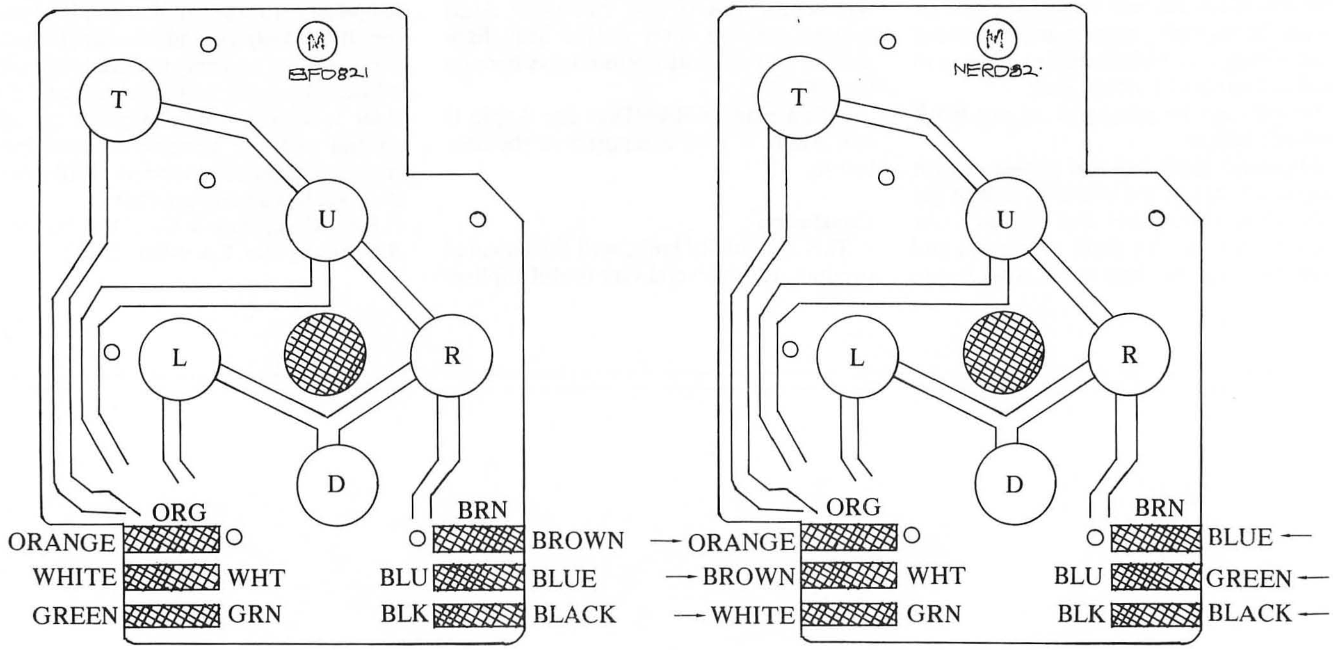
The idea is simple: create a panel of pushbuttons to control all joystick functions. The Asteroids you'll find in arcades, as well as Space Invaders, Galaxian, and other games, use button rather than joystick input. You can open up this realm at home with a few parts readily available at Radio Shack, and the cord from an old stick (you may try finding a DE-9 plug at an electronics store, and making a cord yourself).

You need only a few short snips of wire, some switches, and a box to mount it all in. I used three packages of push button switches (Radio Shack catalog #275-609). These are momentary contact switches, packed two to a package. I mounted five of them in a deluxe project case (Radio Shack #270-222). The total cost of these items was under $8.00, and created a new and enjoyable input device.

Probably the toughest thing about the whole project is putting the mounting holes into the project case. If you don't have access to a drill with a suitably sized bit or hole cutter, you can do what I did: use your soldering iron to start the hole, and then ream it to size using the blade of a scissors. The two tricks to this technique are to work slowly, constantly checking the diameter of the hole against the switch collar, and not burning and/or cutting yourself. It can be done, and that's an

John Anderson is an associate editor for *Creative Computing* magazine.

Figure 1.

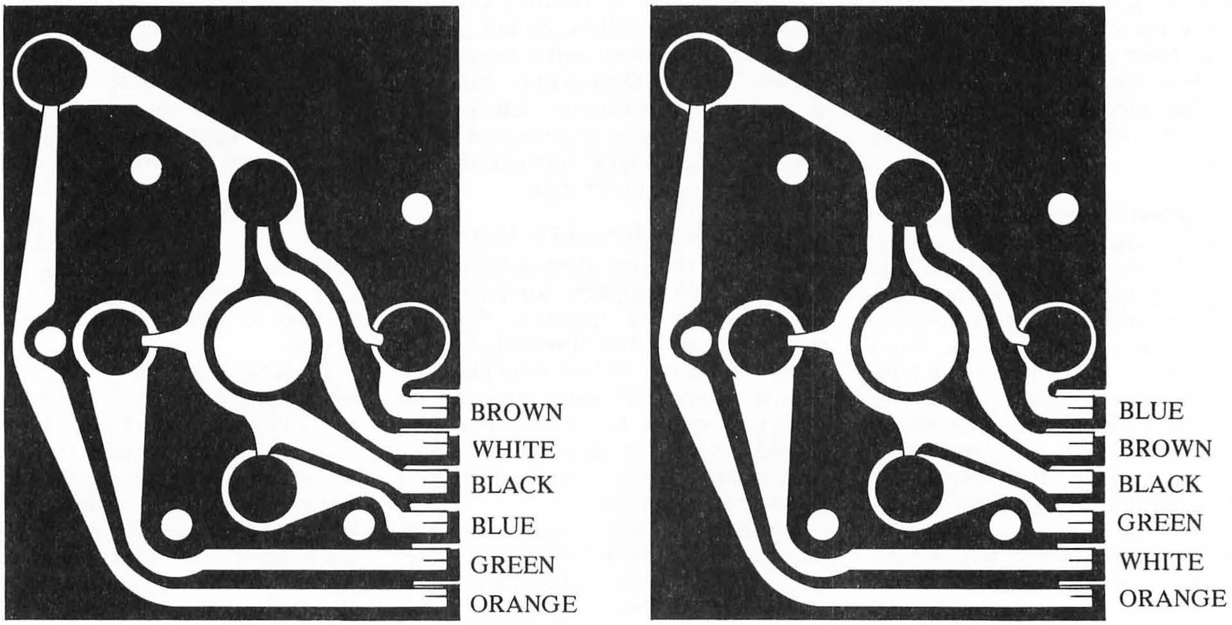


ORIGINAL WIRING

"LEFTY" MODIFICATION

T = Trigger
L = Left
R = Right
D = Down
U = Up

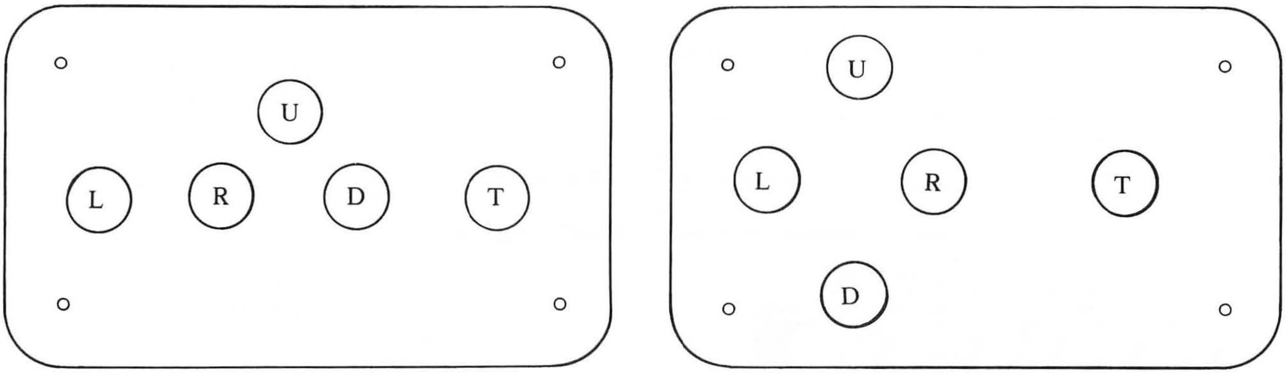
Figure 1A. New Style Joystick.



ORIGINAL WIRING

"LEFTY" MODIFICATION

Figure 2.



"Arcade" Style

"Clock-Directional" Style

T = Trigger
L = Left
R = Right
D = Down (Hyperspace)
U = Up (Thrust)

advantage of a plastic project case (another is its low price).

Refer to Figure 2 for possible button configurations. The first is the "classic" Asteroids format. If you're building a peripheral just to play Asteroids, this is the way to go. The second is what we might call a "clock-directional" format, which in the long run proves to be a more versatile set-up. I made up one of each, and prefer the clock-directional arrangement for a variety of games.

You will need a groove in the box portion of the case to allow the cord to pass through. You may again use the soldering iron to do this, making the groove only wide enough to push the retaining collar in. This way it won't be easy to yank the wire out by its roots.

In order to wire up the new peripheral, refer to Figure 3. As far as I know, this color scheme is standard. In order to attach connectors to the pushbuttons, you'll want to press each connector lightly between the jaws of a pliers. If you are careful about this, you will create a good connection without losing the ability to remove the cable later. Those of you who wish to make your own cord will have to find a DE-9 connector, (which may not be easy), and wire it as shown in Figure 4.

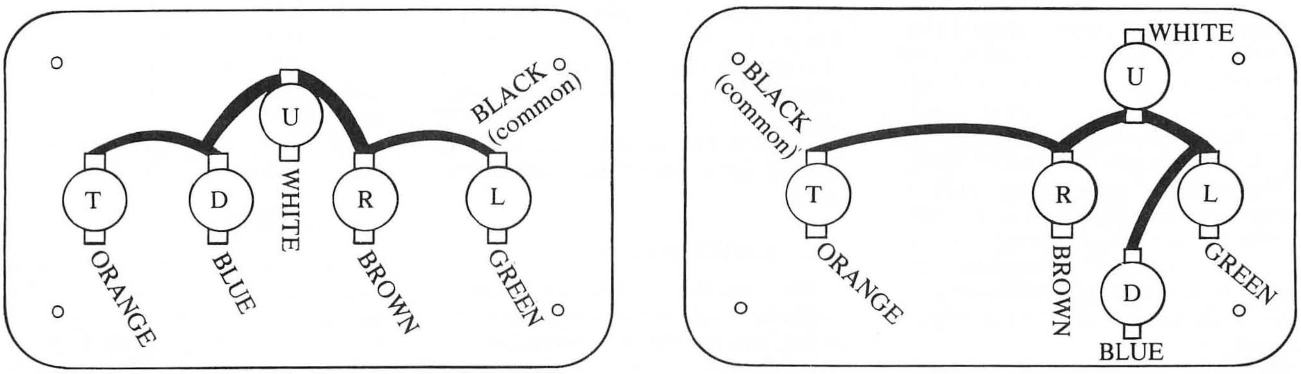
Necessarily, diagonal motion is tough with this configuration, as it requires two buttons to be pressed simultaneously. As a result, games in which the player moves in one dimension are especially suited for pushbutton input (Asteroids is a notable exception). If you feel really brave, try it with a maze game, like *Jawbreaker*.

Double Your Fire Power

If you construct a pushbutton peripheral with the parts I've listed above, you will have an extra button left over. It is a relatively simple matter to attach this button to the handle of an existing joystick, thereby adding a second trigger in a very handy place. It's nice to be able to fire with the same hand that steers, and because the conventional trigger remains enabled, you can easily squeeze off more shots this way.

Use a blade of your trusty (and by this time, quite dull) scissors to press a hole through the top of the stick. Next, disassemble the stick, following the instructions given above for the "lefty" modification. Remove the white plastic stem from inside the handle. Using a saw or serrated kitchen knife, cut off about a

Figure 3. Flip-Side Wiring Diagram.



"Arcade" Style

"Clock-Directional" Style

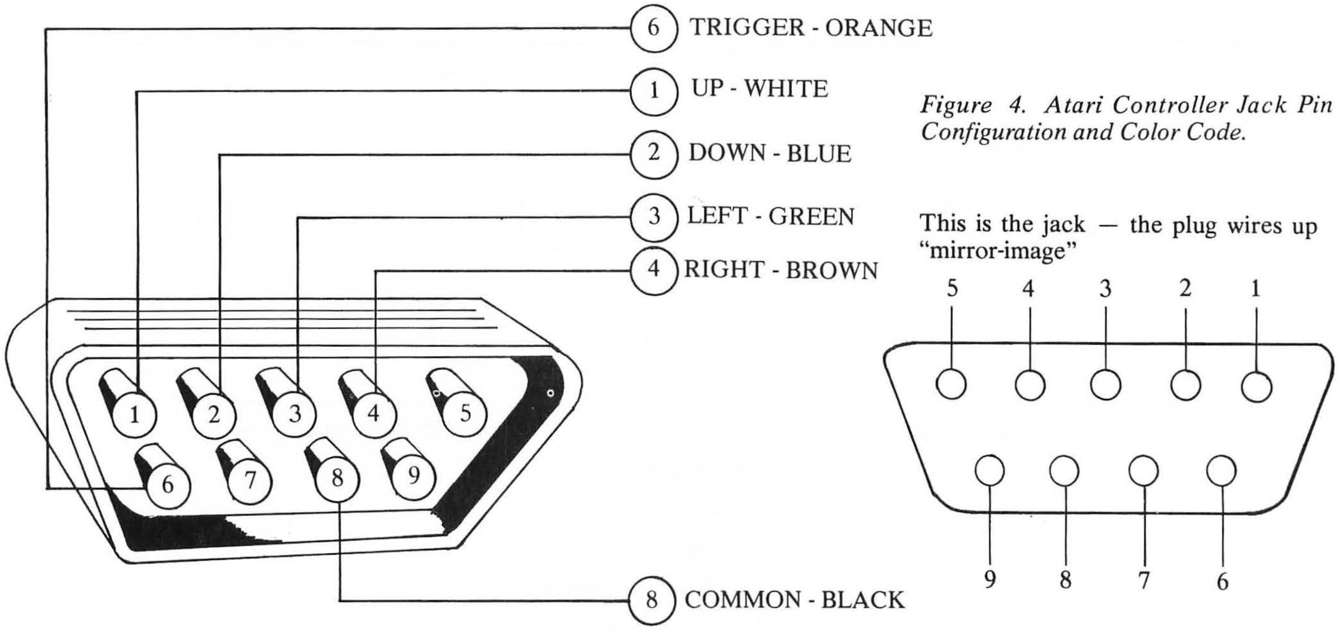


Figure 4. Atari Controller Jack Pin Configuration and Color Code.

This is the jack — the plug wires up "mirror-image"

half an inch from the top of the stem. This will provide the needed room for the switch.

Unscrew all collars and retainers from the neck of the button. Solder two 12-inch lengths of wire to the switch contacts, braiding these leads together. Pass them through the hole you made on top of the stick, and through the white plastic stem. Then screw the pushbutton directly into the top of the joystick handle. The other ends of the leads attach as shown in Figure 5. Reassemble the stick, remaining mindful of that little spring that sits on the original trigger button. You will effectively have doubled your firing ability. Remember, however, some games do not allow for excessively rapid fire play.

Atari Game Controllers

If you like the idea of a pushbutton controller, but lack the time, talent, or inclination to construct one, you may want to purchase one of the ready-made controllers described below.

Starplex Controller

The Starplex controller from Starplex Electronics, offers an authentic "Asteroids-style" button configuration, as well as the fastest set of pushbuttons I have ever seen. In addition, an optional AA battery powers a "rapid-fire" mode, automatically repeating fire faster than you can do it by hand.

Because the pushbutton array is large and has a light touch, the controller takes a bit of getting used to. Eventually, however, I found that the lightning fast direction changes possible with Starplex resulted in higher scores.

It should be mentioned that because many games do not allow a new shot to be fired until an old one leaves the screen, the "rapid-fire" option will not always work optimally. Still, you can fire continuously merely by holding the button down, rather than having to re-press the trigger for each shot (or battery of shots). Over the long haul this reduces fatigue, and the incidence of "joystick elbow."

The unit lists for $29.95, which is a bargain for the most authentic game of Asteroids this side of the coin-op. It improved my score on several other games as well.

Starplex Electronics, Inc., E23301, Liberty Lake, WA 99019. (509) 924-3654.

KY Enterprises

The controller offered by KY Enterprises uses a directional-style configuration, less suitable for Asteroids but more versatile overall. For those unfamiliar with the arcade configuration, it is much easier to master this logical layout.

The unit exhibits extra sturdy construction—as if its makers knew it would have to withstand a few bounces off the floor. It is very large, and can be cradled or used on a tabletop by even the tiniest kids. The buttons themselves sit in raised collars, and, though not as fast as the buttons or the Starplex unit, appear to be the "regulation" coin-op standard. They are large and easy to control.

The KY Enterprises controller is priced at $26.95, and is available in left- or right-handed models. They also manufacture controllers for the handicapped.

KY Enterprises, 3039 East Second St., Long Beach, CA 90803.

Accu-Play

A third pushbutton controller, the Accu-Play Control Board, we did not have an opportunity to test. It sells for $29.95 from Accu-Tech Products, 10572 Swinden Ct., Cincinnati, OH 45241.

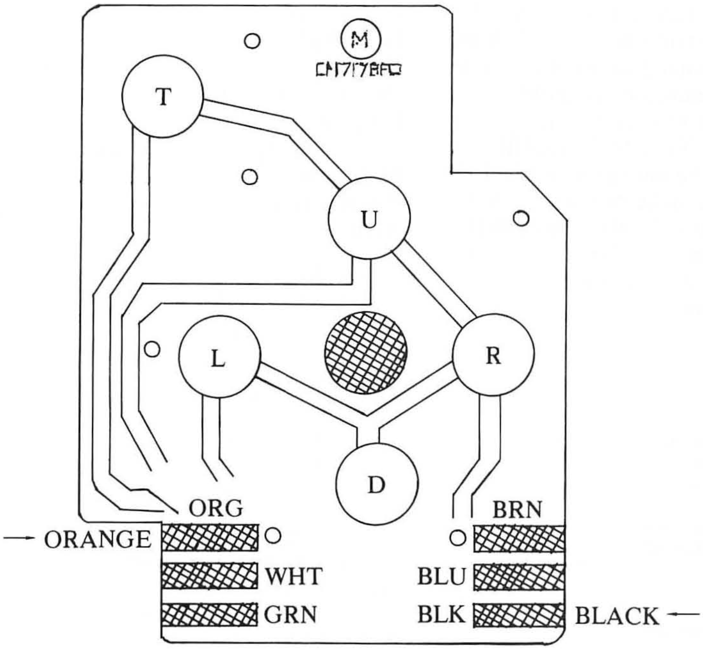


Figure 5. Second Trigger Wiring.

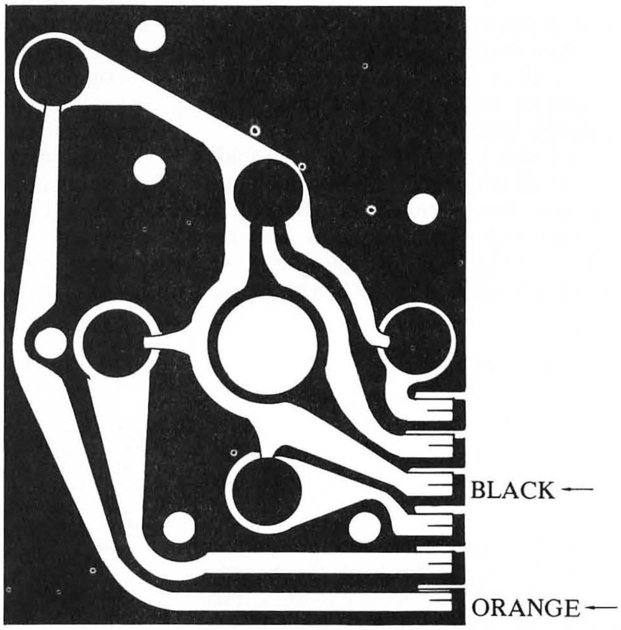


Figure 5A. New Style Stick.

New RAMS for Old

Steve Olsson

The procedure outlined below enables adventuresome Atari users to upgrade 8K memory boards to 16K. While savings of up to $100 per board are possible, users should be aware that this modification voids the warranty on the memory boards.

If you have an Atari 800 with two 8K memory boards and don't want to upgrade memory by throwing away two expensive modules, you can now upgrade them to two 16Ks for a fraction of the cost of new 16K boards. This upgrade can be done by almost anyone, and does not require extensive hardware knowledge. All it takes is a bit of soldering. The theory is as follows:

The 4116 dynamic memory is a very popular memory chip used by, among others, Apple, TRS-80, and Atari. This chip is inexpensive and readily available. It is arranged as a 16K x 1 in a sixteen pin DIP and comes in many different speeds.

Steve Olsson, 3392 Clipper Dr., Chino, CA 91710.

The 4116 memory also has a half brother, the 4108. The 4108 is very similar to the 4116, except it is arranged as an 8K x 1. In reality, the 4108 chip is a 4116. Besides the label, there is only one real difference: the 4108 is a 4116 that has a problem. When the chips are manufactured, bad ones are thrown into the reject pile and good ones are shipped. From the reject pile some chips are again sorted and shipped. Chips with the upper half bad and lower half good are sold as 4108-A, and those with the upper half good are sold as 4108-B.

Atari now buys a 4108 chip and accesses only the good half of it on the 8K memory board. If Atari were to install completely good 4116 memory chips and access the entire chip, a 16K memory board would result.

The point is, instead of throwing away the 8K module (which is nearly identical to the 16K module), why not replace the 8K memory chips with 16K memory chips? Several jumper options must be changed, and the 8K memory must be removed from its sockets and replaced with 4116s. The whole process is extremely easy and should take about 30 minutes.

In order to begin the procedure, the first thing to do is order eight 4116 RAMS per board being upgraded from a local supply house. (Care must be taken to choose a reputable supplier. The parts should be guaranteed 100% operational). The cost of the chips ranges from $30 to $60. The chips must to have a maximum access time of 200 nS in order to work in the Atari.

Once the 4116s are in hand, open the top of the Atari and remove an 8K memory module. Remove the two screws that hold the memory module together. Pop off the metal cover and snap open the module along the edge connector. The circuit board now lifts out of the module.

Six jumpers on the front (component side) of the board labeled A, B, C, D, E, F are now exposed. They are actually resistors of very low value but function as jumpers only.

The edge connector is labeled 1-22 on the front and A-Z on the back. (Notice omitted letters G, O, Q, I due to similarities in shape.) The letters connected together by small pieces of etch are: U-T, S-R, and N-P. Also notice the etch from W to Z501 pin 15. All of these small etches must be completely removed with a razor blade or X-acto knife.

Atari was nice enough to add solder holes to all of the connections which must now be soldered. Connectors to be soldered together with small pieces of wire are: Z501 pin 15-U, T-S, P-R, and M-N.

On the front side of the board, jumper C must be installed and all other jumpers removed. On the back of the board a very small solder connection must be made to the connector H as far away from the edge as possible. This wire must be added to hook that signal to jumper D on the side next to the letter (as shown in Figure 4). Make this connection from the back of the board even though the letter is on the front of the board.

Program 1.

```
10  GRAPHICS 8
20  SETCOLOR 2,0,0
30  COLOR 1
40  FOR Y=0 TO 159
50  PLOT 0,Y
60  DRAW TO 319,Y
70  NEXT Y
```

The next step is to remove the 8 DIPs labeled C503, C505, C507, C509, C511, C513, C515, and C517 from their sockets and replace them with the 4116s. Replace the board in the module, screw it back together, and the modification is finished!

In order to test the memory, use the following procedures: Insert only the module under test into the Atari then use the ?FRE(0) command to see if the Atari recognizes an increase in memory. If everything looks OK at this point, use graphics 8 mode. Type SETCOLOR 2, 0, 0, which makes the background black. If no spots appear, make the screen white by using Program 1. If, after running this program, there are no holes in the screen pattern, assume the last 8K of memory has no solid errors.

Program 2.

```
  2  X1=14*256
  4  X2=65*256
  6  X=14
 10  POKE 106,X:GRAPHICS 0
 20  FOR X=X1 TO X2
 30  POKE X,.255
 40  NEXT X
 45  FOR X=X1 to X2
 50  IF PEEK (X)<>255 THEN PRINT
"ERR-";X
 60  POKE X,0
 70  NEXT X
 80  FOR X=X1 TO X2
 90  IF PEEK (X)<>0 THEN PRINT
"ERR-";X
100  NEXT X
```

After this test run Program 2 to check more of the memory. This program checks each memory location (without interfering with Basic) and reports failures to the screen. A few failures could mean there are some bad chips; many failures probably mean the module was wired wrong or the chips are very bad. The failure will probably have to be determined from the failure report generated by Program 2, which reports the address of failure. PEEK and POKE must be used to determine which bit is bad. Program 2 cannot check the first 5K of memory in the module, but if the program runs without strange things happening it is probably all right.

If a memory board is known to be good, place it in slot 1 in memory. If the total memory is now 24K, change lines in Program 2 to:

```
2 X1 = 32*256
4 X2 = 96*256
6 X  = 32
```

If the total memory is 32K, change lines in Program 2 to:

```
2 X1 = 64*256
4 X2 =128*256
6 X  = 64
```

The program can now be run. This will completely test the new memory module, and will take about 10-14 minutes to run. If you had only one 8K module that is now a sixteen, you will have to hope the first 5K of memory is good until you get more. The first 5K is impossible to test with only one module.

If your computer passes all these tests, the memory in your Atari has just been doubled. If you have any trouble that is not understandable and have rechecked the procedure to verify that it was done right, you probably have bad RAMs.

This simple procedure will, I hope, save many people lots of money, allowing them to operate with a disk drive and have plenty of memory left for the other programs. □

Photo 1. The open 8K module.

Photo 2. The component side of the memory board.

Photo 3. Close up of the jumpers A-F.

Photo 4. Correctly installed 16K jumpers on back of board.

Photo 5. The etch side of the completed mod.

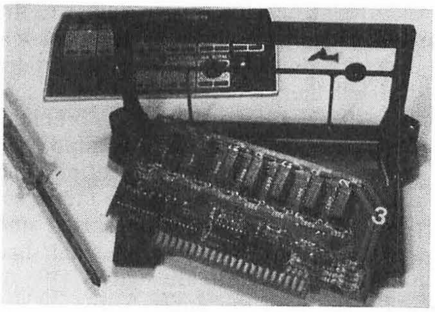


Photo 1.

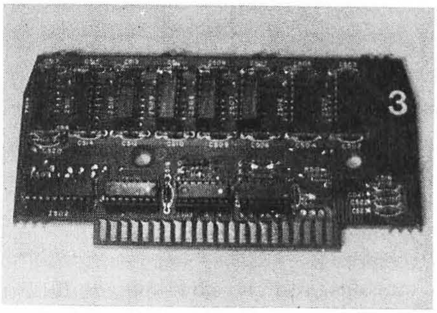


Photo 2.

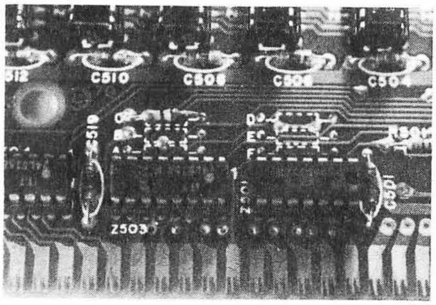


Photo 3.

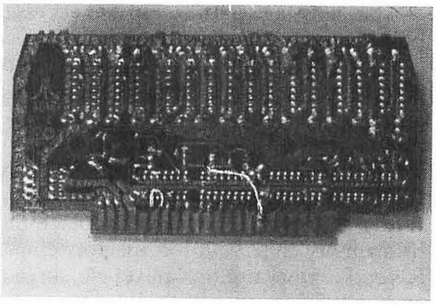


Photo 4.

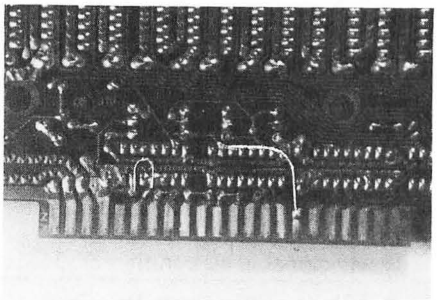


Photo 5.

K-DOS — An Alternative to Atari DOS

Sheldon Leemon

K-DOS from K-Byte is an alternative to the Atari Disk Operating System, DOS II. The file management system of *K-DOS* is compatible with Atari DOS, but offers a greater level of control over peripheral devices and memory. Although it offers many features which will be appreciated by every Atari user, *K-DOS* will be of most use to the serious programmer.

Since the benefits to be gained by using *K-DOS* are the result of certain trade-offs, the potential buyer should think hard about how much a more convenient disk operating system is really worth.

Chief among these trade-offs is the amount of memory that *K-DOS* leaves available to the user. *K-DOS* is memory resident, so most of its features are immediately accessible, but it also takes up a great deal of space. With a Basic cartridge inserted, the amount of free memory available in a 40K system is 25,228 bytes. This is almost 7K less than the 32,274 bytes available with Atari DOS, or the 31,758 bytes available with OS/A+.

Besides reducing the amount of memory available for programming, the large size of *K-DOS* puts the start of low memory above $3000. (An optional program included with the package lets you remove the plain English error messages, which saves enough bytes to bring the end of *K-DOS* just below $3000). Machine language programs which are assembled to run just above the end of Atari DOS, may conflict with *K-DOS*, and may not run under it.

If you have memory to spare, however, *K-DOS* offers many attractive features. It is, for example, a pleasure to use: all DOS functions are accessible from Basic, Pilot, the Assembler cartridge, or whatever program environment you happen to be in.

Since *K-DOS* is command driven, you need not call up a menu to execute a DOS function. You simply precede the command by a comma (or some other character which you can define as significant to DOS), and the DOS function is executed without changing program environments.

The syntax required for command lines is flexible, so commas can be replaced by spaces, lower case is acceptable, and the DOS environment automatically resets the inverse character shift. Device defaults are supplied whenever possible, and short abbreviations are allowed, so a minimum of keystrokes is required to perform any function. Error messages appear in plain English, rather than a frustrating number code.

Unlike OS/A+, which puts you back in the operating system every time you hit System Reset, *K-DOS* will only bypass Basic if you hit the Start key along with System Reset. And unlike Atari DOS, the device handler for the 850 interface unit boots automatically if it is turned on. There is no need for a separate AUTORUN.SYS file.

The reason that *K-DOS* can let you use DOS command lines from Basic is that it re-routes all input to the line editor (although it gives you a command, KILL, which will take its "hooks" out of the handler table if desired).

This greater level of control over the system is characteristic of *K-DOS*. For example, the 6502 BREAK instruction is vectored to get you back to DOS any time the instruction is encountered, rather than having the system hang up. You may get a little better idea of what this means if you slip in the Basic cartridge and type INPUT (RETURN).

With Atari DOS II, the system locks up, and the only way to recover is to turn the computer off and reboot. With *K-DOS*, a BRK message appears, and you enter DOS. You should even be able to recover from the dreaded "editing lock-up," which occurs when Basic moves a block of exactly 256 bytes (You must still know enough about how Basic works to reset the statement pointers, however, as that particular bug tampers with your program code before it crashes the system).

Another aspect of the system control offered by *K-DOS* is that it allows you to stop disk I/O just by hitting the BREAK key, without destroying your data. It also tries very hard to read and write marginal sectors before bombing out, which is important, given the notorious speed fluctuation of older Atari disk drives.

SOFTWARE PROFILE

Name: K-Dos

Type: Operating system

System: Atari 400/800, 48K preferable

Format: Disk

Language: Machine

Summary: Versatile, but memory-hungry alternative to Atari DOS

Price: $89.95

Manufacturer:
K-Byte
P.O. Box 456
1705 Austin
Troy, MI 48099

K-DOS puts some nice touches on some of the original DOS functions. For example, INIT combines formatting and writing DOS files to the new disk in one operation, although these functions are still available separately. The duplicate disk function offers the option of a straight sector copy for boot-disks that do not have file information on them, and also allows the faster write without verify and continuous retrying of bad sectors.

There is a separate APPEND command, which allows you to enter data at the end of a file directly from the keyboard. The append function uses any space available in the last sector, rather than starting a new sector as Atari DOS does. The binary load command prints to the screen the location in memory into which the file is being loaded, if you so desire, which is much more convenient than reading the headers and calculating the addresses by yourself.

But *K-DOS* doesn't take up all that memory for the sake of a few slight modifications. It also contains a complete machine language monitor which allows you to examine memory in hexadecimal and ASCII formats, alter memory by typing in either hex or ASCII values, and examine and alter the contents of the registers. *K-DOS* gives you two ways to execute a machine language program. GO runs the program after closing all devices, and does not preserve the registers. PROCEED continues a program after a breakpoint has been reached, without changing the contents of the registers or the status of any device, making it a very handy debugging tool.

Similarly, the command XIT allows you to get back to a Basic program that calls DOS, and continues to run that program from the point at which DOS was called.

A null device handler has been added, so that you can test I/O operations quickly by directing them to N:. LOMEM lets you examine and alter the bottom of memory available to a cartridge. This allows you to reserve space for machine language programs, or just to reduce the amount of memory available to see if a Basic program will run on the minimum 16K system. UDC allows you to add your own user-defined commands to the system.

In addition, *K-DOS* offers many commands which allow you to access certain routines used internally by DOS, just by giving a one-word command. For example, COLD and WARM provide an easy way to coldstart or warmstart a

Sheldon Leemon, 14400 Elm St., Oak Park, MI 48237.

cartridge. RESET reboots the 850 handler when you have expanded the drive buffers —or just forgotten to turn it on when you booted up.

TEXT corresponds to a GRAPHICS 0 call in Basic, and opens the screen device, which is handy for moving the display list when you want to load a program into high memory. CLOSE closes all files, turns off the sound, resets VBLANK vectors, and turns off Player-Missile graphics. ER followed by a number will print the English error message for that error number, which is very handy when you want to interpret I/O errors that are generated by Basic.

None of these functions is earth-shaking, and all can be accomplished in other ways with a little effort, but the author's attitude was that as long as the routines for doing them were already in DOS, it made sense to allow them to be accessed easily.

Unfortunately, the lack of depth in the documentation runs somewhat counter to this intention of allowing the programmer easy access. The glossy K-DOS Handbook is nicely bound, comes with a pocket summary card, is clearly written, gives examples of the proper syntax for each command, and covers most of the commands very well.

However, it treats some of the more esoteric commands in a cursory manner. Take, for example, the explanation of the UNLOAD command: "Tries to erase area where cartridge is; unloads any RAM based cartridge and resets LOMEM back to end of DOS." The beginner will no doubt read this sentence, re-read it once to verify that all of the words are in English, and then press on, no better or worse for the experience.

The experienced user, on the other hand, might gather from this explanation that it is possible to load a program into RAM, and fool the system into thinking that the program is cartridge-based, allowing an easy transition back and forth between that program environment and DOS. The inference would then be that the UNLOAD command erases this program, and lets the system know that no cartridge is present. But how do you set up this "RAM based cartridge" in the first place? No clue is given, leaving the experienced user perhaps more frustrated than the beginner.

Another example of a similar sort is the system equate files that are supposed to give the user access to system routines, such as the one to type text messages from a buffer. There are no detailed examples of how to use them, however, and the internal commenting is too scanty to allow most users to benefit from them. Features like these could be real selling points to the ambitious programmer if they were treated less superficially in the documentation.

My impression of *K-DOS* is that aside from these omissions in the documentation, it is a convenient tool for the user who is serious about programming.

As one who uses his computer mostly for programming, I have found *K-DOS* especially helpful in developing software that combines Basic with machine language subroutines. But I think that *K-DOS* will be of much less interest to the casual programmer who may have less than 40K of memory.

While such a user might appreciate some of the features, he would probably never take advantage of the machine language monitor, the null device, or many of the other goodies which make *K-DOS* so big—and so expensive. If you fall into that category, you might be better off spending the money on something that will let you gobble dots, eradicate insects, or save the universe. □

Standard Keyboard for the Atari 400

Robert Noskowicz

While shopping for a home computer, I did quite a bit of research, eventually narrowing my decision to a choice between the Atari 400 and 800. With a little more investigation I found that the only differences between the two are the three most obvious: 1) easy access to additional memory, 2) the two ROM slots and 3) the keyboard. The processors—operating systems and ROM—are exactly alike. Since Atari still has not used the second ROM slot, and the 400 can be fairly easily upgraded to 48K, the only appreciable difference is the keyboard. The 400 has a flat membrane keyboard compared to the standard typewriter keyboard on the 800. I didn't feel at that point that the differences warranted the approximate $400

Robert Noskowicz, 44 York St., Old Bridge, NJ 08857.

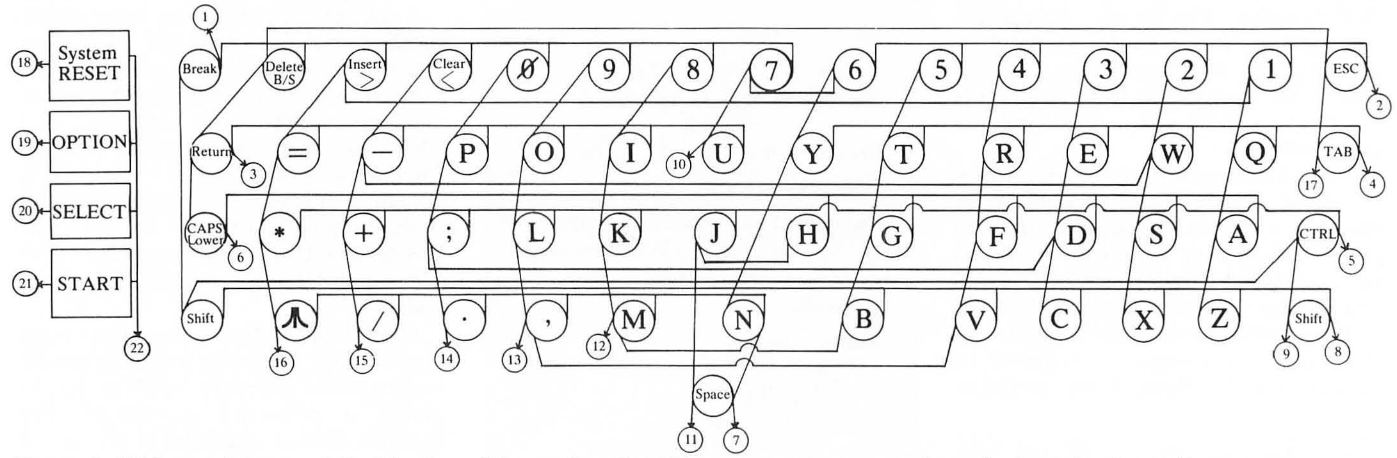


Figure 1. Ribbon cable on original keyboard is numbered 1-22. Keyboard is viewed from the back in above diagram.

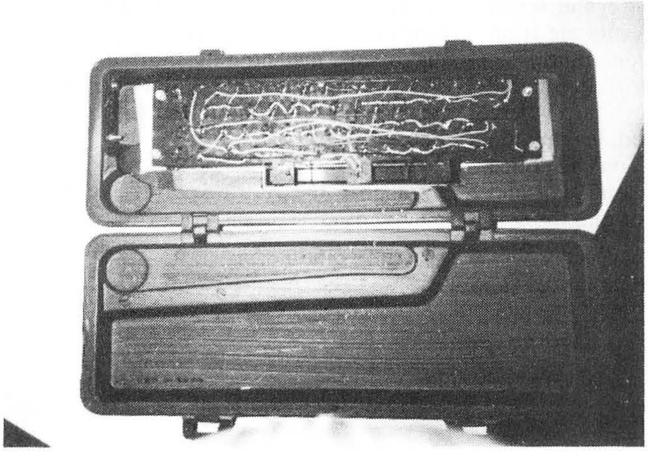


Photo 1.

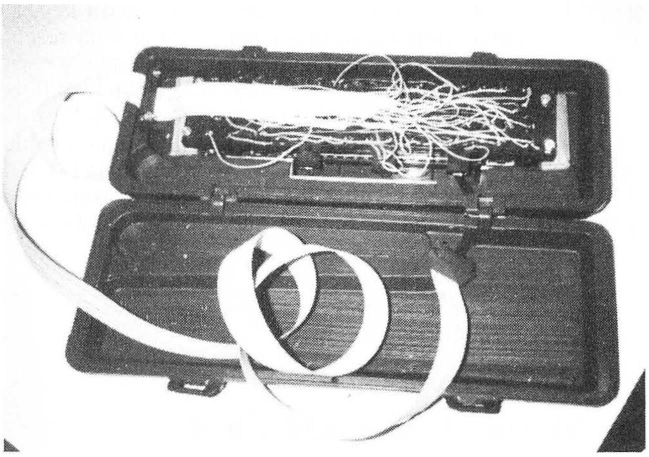


Photo 2.

additional cost for the 800, so I purchased the 400.

After 6 months of use, I was extremely happy with my computer except for the keyboard. I found that the flat keyboard impairs the ability to enter data quickly as well as causing discomfort when entering a substantial amount of information into the system. I went from "I'll get used to it" to "It's not all that bad" to total exasperation.

What I will explain here is what I did to cure my problem: I added a standard keyboard to my machine.

First I opened my computer to determine how the keyboard was interfaced. I had the Atari Technical User's Notes but they did not contain any schematics for the keyboard. After calling several home computer stores to see if they had any information on changing keyboards, with no luck, I called Atari's toll free number in California. If you have ever called Atari, you already know that (like most computer manufacturers) they do not like you to make changes in their hardware and provide very little technical help.

I realized that I would have to do everything myself. The one thing I did know was that the decoding of the keyboard is done in the processor. The keyboard, in the case of the Atari, is just a bunch of momentary ON switches, 61 to be exact. I sat down with my ohm meter, went from point to point, and drew the keyboard layout (Figure 1).

The next step was to purchase the necessary parts. The keyboard that I bought from a firm in California has 62 keys and costs $35. It is called a bare keyboard because it is not mounted on a PC board. Initially I intended to mount the keyboard on the computer but my wife suggested that I use a cable and keep it separate.

This was an excellent idea since I keep the computer on a Parsons table in front of my TV and sitting on the couch slouching over it can be a real pain in the neck, literally. Now I can keep the keyboard on my lap which I find extremely comfortable.

If you decide to go this route, you will need about seven feet of ribbon cable which costs approximately 60 cents per foot. I used 25-conductor cable because I wanted to have a connector between the computer and the new keyboard so that I would be able to disconnect it. Otherwise it would only require 22-conductor.

The connector is a 25-pin RS-232 type made for ribbon cable. It costs about $14 per set. If you want a case for the keyboard, you can purchase one for about $56. If you are like me and wish to save some money, go to your local hardware store and buy a small Permanex tool box which costs about $6 and cut it to shape.

Some of the keys on the new keyboard

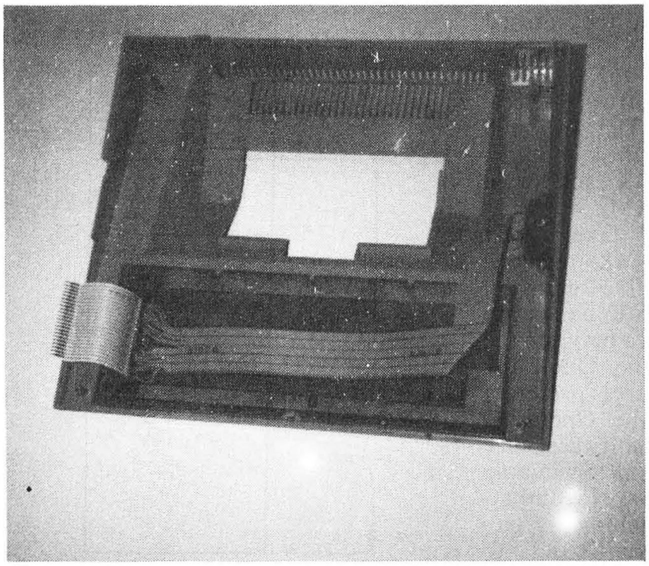


Photo 3.

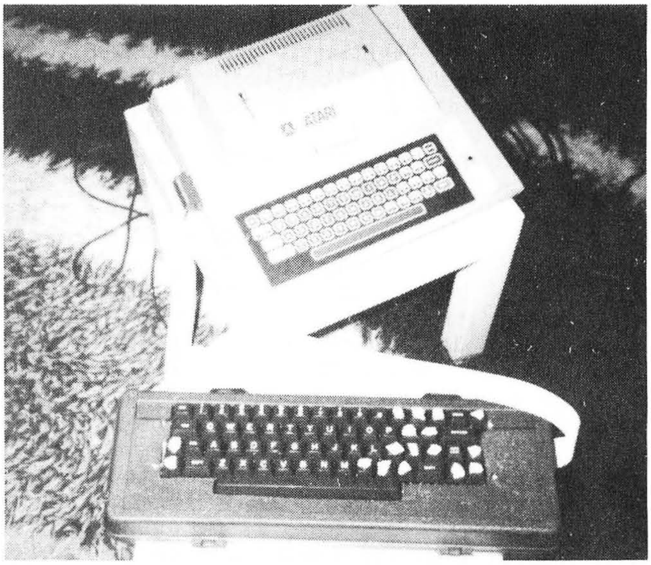


Photo 4.

are in different locations. You can leave them as is or move them about, providing you follow the wiring layout in Figure 1.

One thing I had to do on the new keyboard was to keep the Cap/Lock key from locking, since on the Atari the Cap Key does not lock.

The first step was to wire up the new keyboard (Photos 1 and 2). Since I had one extra key, I used two keys in series for system reset. This prevents me from accidentally resetting my computer.

Next I soldered the new ribbon cable to the back of the Atari keyboard where the original cable is soldered, leaving the original in place (Photo 3). You will notice that I routed the cable to one side and mounted the male connector into the side of the casing. I then assembled the computer and tested the Atari keyboard to make sure that nothing shorted. So far, everything tested OK. I then plugged my new keyboard in and tested it. It worked fine.

The keys on the new keyboard are parallel to the Atari, so either keyboard can be used. □

The Mosaic 64K RAM Card

Atari Supercharge

LeRoy J. Baxter

I found the prospect of dismantling my Atari 800 to install the Mosaic Select (64 K memory board) a bit frightening since I had never been inside a computer before. As it turned out, however, I shouldn't have worried. The computer won't fall apart just by breathing on it, and it is not really any more delicate than a stereo or calculator.

My new memory board was actually designed for the Atari 400, but had been especially modified (components added) to work in the 800. Even though the installation manual was written for the 400, it was clear and complete enough to be of great help in modifying the 800. (By the time you read this, Mosaic should have the memory board for the 800 available and you won't have to translate instructions from the 400 manual.)

Once the computer is disassembled, only two modifications need to be made:

First, a two-wire cable needs to be soldered to the main board. It may be best to use a pencil-type soldering iron with a very small tip. If you have not soldered a printed circuit board before, this is the one step you may want someone else to do.

The second modification requires relocating one of the computer chips from the main board to the new memory board and installing a preassembled flat ribbon jumper cable from the socket on the main board to the memory board.

The rest of the job is just a matter of reassembling the computer in reverse order from its disassembly — a task that can be done in one evening and, with a little practice, could probably be done in less than a half an hour.

Mosaic uses only the best components and gives an amazing four year guarantee that is not limited by a lot of hedges and/or disclaimers.

Just Another Memory Board?

The Mosaic 64K Select is memory expansion with a difference. The diagram in Figure 1 tells the story.

First, Select expands the RAM of your Atari 400 to the design maximum of 48K and then goes on to give you 4K more RAM located in the unused ROM area. Further, this 4K of additional RAM is really 16K — it is addressable as four software-selectable banks of 4K each. The Atari 800 can support three of these boards, giving you 32 banks of 4K each for an astounding 192K of RAM.

Further, Mosaic has taken great pains to make their 64K memory board totally compatible with all existing software. The 4K banks are placed in the unused area between the Basic cartridge and the Operating System ROM — an area presently untouched by Basic, the Operating System, DOS, or any software. The method chosen for Bank Switching also precludes any software incompatability as Bank Switching is accomplished by writing (ie. POKE) to ROM. Since the ROM areas are cast in stone (silicon), as it were, nothing is actually written — it is the act of writing that is important. The specific address that you try to write to determines the Bank that will be selected (see Listing 1).

Note that while the Mosaic 64K Select is totally software compatible, it is not compatible with certain hardware modifications such as the 80-column board.

With the Basic cartridge removed, machine-language programs such as the Atari Word Processor or Visicalc see 52K of continuous RAM — a big boost in available RAM.

With the Basic cartridge installed (or Microsoft loaded), the normal

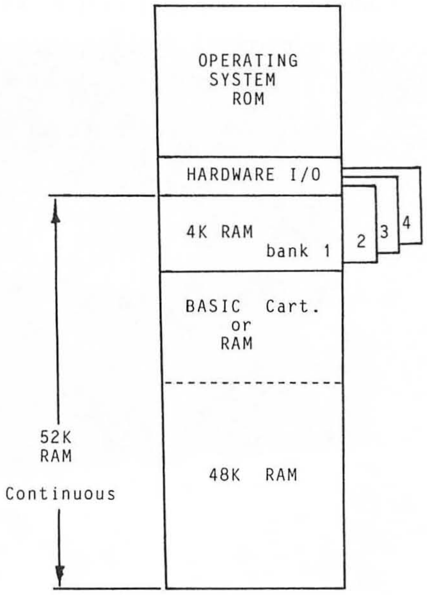


Figure 1.

LeRoy J. Baxter, 15601 S.E. Oatfield Rd., Milwaukee, OR 97222.

40K maximum RAM is available, along with the four 4K banks that Basic can't see (at least not without help).

What good is "invisible" memory? One of the big problems with using machine-language routines with Basic has been finding a safe place to put them. Until now, page 6 (memory starting at 1536) has been a popular place. Too popular in fact, and a dangerous place since under certain conditions, cassette input will use the bottom half of page 6 as a buffer. Any machine-language routine stored there will be lost. This can be especially damaging if the routine is a Vertical Blank routine.

In contrast, machine-language routines or data stored in the Select Banked Memory areas are 100% safe. An entirely new program may be loaded without affecting anything stored there. In addition, a lot more room is available — a whopping 4096 bytes instead of just 256 bytes for page 6.

If you have a Mosaic Select, try this:

```
10 POKE 65472+0,0:REM SELECT BANK 0
20 DIM A$(19):A$="this is a LOAD test"
30 FOR X=1 TO 19:POKE 49152+X,ASC(A$(X,X)):NEXT X
```

then type:

RUN
NEW
LOAD "D:filename" (load any program) or load a program from cassette

finally (in Direct Mode):

```
FOR X=1 TO 19:? CHR$(PEEK(49152)+X;:NEXT X
```

Pressing System Reset does not affect the stored data. Only turning off the computer or overwriting the data will destroy it.

Player/Missiles and Character Sets

With normal memory management, finding a suitable place for Player/Missile data and/or redefined Character Sets can be a problem. Care must be used to position PMBASE and/or CHBAS on the appropriate 1K or 2K boundary. If the memory area is reserved by moving RAMTOP, then care must also be used to prevent the Display Memory from crossing a 4K boundary. The pitfalls are many and often large blocks of memory end up unused.

Other Applications

The Mosaic Select is in keeping with the open-ended, nature of the Atari itself. Many additional applications come to mind:

1) Relocate the String-Array space to the banked memory area. This could be useful for chained Adventure programs or for Financial/Budget applications that do different things with the same data. The String-Array data would be instantly available to each program as soon as it was loaded and run.

2) Different sets of data could be loaded into each of the four banks, letting one program act like four. Each set of data would be instantly available with a single POKE.

3) The possibility exists that relatively short Basic programs (less than 4K each) could be stored in each bank, allowing four totally independent (and one Master) programs to be in memory at once and available with just a few POKEs.

4) Machine-language programming Utilities, DOS Utilities, and/or Wedges could be stored in these banks. These utility programs would be easily accessed but would not be affected by LOADs, SAVEs, or RUNs. Nor would they ever conflict with the Basic program area or the Display List/Display Memory. Different utilities could be stored in each bank and accessed when needed.

5) For Assembly-language programmers using the Atari Assembler-

Listing 1.

```
5 REM "YOUR NAME IN LIGHTS" (C) 1982 by LeRoy J. Baxter
6 REM IF TOO MUCH FLICKER, ADJUST TV BRIGHTNESS AND CONTRAST
10 POKE 106,207:REM RAISE RAMTOP
20 BNKBAS=65472:BANK=0:BANKSELECT=2000
30 POKE BNKBAS+1,BANK:GRAPHICS 5+16
35 REM GET START ADDR OF GR.5 DISPLAY LIST
40 AL=PEEK(560):AH=PEEK(561):AD=AL+256*AH
50 POKE BNKBAS+2,BANK:GRAPHICS 5+16
60 POKE BNKBAS+3,BANK:GRAPHICS 2+16:B=PEEK(560)+256*PEEK(561)
70 POSITION 8,5:? #6;"your":POSITION 8,6:? #6;"name"
75 REM MOVE GR.2 DISPLAY LIST TO START AT SAME LOCATION AS GR.5 LIST
80 FOR X=0 TO 18:POKE AD+X,PEEK(B+X)
90 IF PEEK(B+X)=65 THEN POKE AD+X+1,AL:POKE AD+X+2,AH:X=18
100 NEXT X
110 POKE BNKBAS+0,BANK:GRAPHICS 5+16:POKE 710,154
115 REM NOW DRAW TO GR.5 SCREENS - 1/3 OF DATA TO EACH BANK
120 FOR X=0 TO 15:READ A:POKE 1750+X,A:NEXT X
130 FOR X=0 TO 10:READ A:POKE 1710+X,A:NEXT X
140 FOR X=0 TO 2:POKE BNKBAS+X,X:COLOR X+1:PLOT X*2,19:DRAWTO X*2,28
150 PLOT 79-X*2,19:DRAWTO 79-X*2,28:NEXT X
160 FOR Z=0 TO 2:COLOR Z+1:BANK=Z
170 FOR X=39 TO Z STEP -1:GOSUB BANKSELECT:PLOT 39-X,18-Z:PLOT 40+X,18-Z:PLOT 39
-X,29+Z:PLOT 40+X,29+Z:NEXT X
180 READ A:FOR X=1 TO A:GOSUB BANKSELECT:PLOT 39-Z,18-X-Z:PLOT 40+Z,18-X-Z:PLOT
39-Z,29+X+Z:PLOT 40+Z,29+X+Z:NEXT X
190 FOR X=1 TO 4:GOSUB BANKSELECT:PLOT 39-X-Z,18-A-Z:PLOT 40+X+Z,18-A-Z:PLOT 39-
X-Z,29+A+Z:PLOT 40+X+Z,29+A+Z:NEXT X
200 READ A:FOR B=1 TO A:READ X,Y:GOSUB BANKSELECT
210 PLOT 39-X,18-Y:PLOT 40+X,18-Y:PLOT 39-X,29+Y:PLOT 40+X,29+Y:NEXT B
220 NEXT Z
230 FOR X=0 TO 2:POKE BNKBAS+X,X:POKE 49152,3:NEXT X:REM SHADOW REG. FOR BANK #
240 X=USR(1710):REM START BANK FLIPPING
250 FOR X=0 TO 2:POKE 1700,X:FOR T=1 TO 20:NEXT T:NEXT X:GOTO 250
280 REM PAGE FLIP ROUTINE
290 DATA 174,0,192,224,3,240,3,174,164,6,157,192,255,76,95;228
300 REM SET VBLANK ROUTINE
310 DATA 104,162,6,160,214,169,6,32,92,228,96
320 REM SCREEN DATA
340 DATA 10,24,5,11,6,11,7,12,8,12,9,13,10,14,10,15,9,16,8,17,7,17,6,17,5,17,4,1
7,3,16
350 DATA 2,16,1,15,1,14,2,13,3,13,4,13,5,14,6,14,6,15,5,15
360 DATA 6,37,6,8,7,8,8,9,9,9,10,10,11,11,12,12,13,13,14,14,15,15,16,15,17,16,18
,16,19,17,20,17,21,17
370 DATA 22,16,23,16,24,15,25,14,25,13,24,12,23,12,22,11,21,11,20,10,19,10,18,10
,17,11,16,12,17,13
380 DATA 18,14,19,14,20,15,21,14,20,13,20,14
390 DATA 2,54,7,5,8,5,9,5,10,5,11,6,12,6,13,6,14,6,15,6,16,7,17,7,18,7,19,7,20,8
,21,8,22,8,23,8
400 DATA 24,9,25,9,26,9,27,9,28,9,29,10,30,10,31,10,32,10,33,9,34,9,35,8,36,8,37
,7,37,6
410 DATA 36,5,35,5,34,5,33,4,32,4,31,4,30,4,29,4,28,5,27,5,26,5,25,6,26,7,27,7,2
8,7,29,7
420 DATA 30,8,31,8,32,7,32,6,31,6,31,7
2000 POKE BNKBAS+BANK,BANK:BANK=BANK+(BANK<3)-3*(BANK=2):RETURN
```

Editor, machine-language routines could be written and assembled in the memory area where they will reside — a lot easier than writing relocatable code.

6) The fact that each bank is selected by a single POKE allows various forms of "Page Flipping." The program in Listing 1 demonstrates how this can be done. A separate display is placed into each bank and then the banks are flipped using a short machine-language Vertical Blank routine. This kind of page flipping allows color blending and the mixing of text and graphics.

Note that with 4K in each bank, Graphics 6 is the highest resolution mode that can be used. A Graphics 7 Display Memory can fit into 4K, but not both the Display List and the Display Memory.

Also note that if overlaying different graphics modes, either the Display Lists must start at exactly the same memory location, or the Display Lists must be chained.

As with the Atari itself, the list of applications goes on, limited only by the programmer's ability and imagination.

The Future

The first practical applications will probably be Programming Utilities and Wedges that can be relocated to lower memory areas if Select has not been installed. The 64K board will mean that more of these utilities can be in memory and available to the programmer at any one time with less interference with the main program.

The second most practical application might be for Player/Missiles and Character Set data. By simply writing to a location in this area and then reading that same location, the existence of 64K can be determined.

```
POKE 49152,3:IF PEEK(49152)=3
     THEN PMBAS=49152/256: etc.
```

Assembly-language programs can check against RAMTOP. If RAMTOP is greater than 192, then 64K is installed.

I am sure that the future will see commercial programs written that will test for the existence of 64K RAM and will load in more data for bigger and better programs if it is installed.

The Mosaic 64K RAM Select appears to be an innovation whose time has come.

New Member of the Family

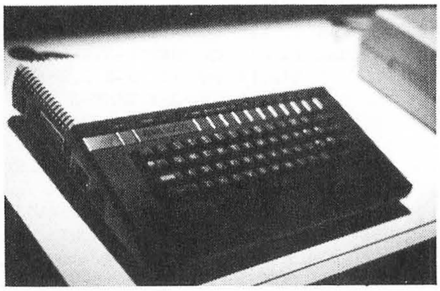
Atari 1200

John Anderson

Well you may or may not have heard the news, but the Atari 1200 has arrived. Here is a first look at the 1200XL, and the new wave of peripherals and software designed to work with it.

The Atari 1200XL was unveiled on the east coast at a press conference at the Plaza Hotel in New York City. At least a dozen working units were on display there for us microcomputer types to play with, and that's exactly what we did, (at great length). The unit should become generally available by the middle of 1983.

With 64K RAM standard, the 1200XL also offers twelve user programmable function keys, international character set, and built-in diagnostics. Designed to be entirely compatible with the models 400 and 800, owners of the Atari 1200 need not, therefore, have to wait for software to be developed to run on their machines. Although no true innovations are present in the 1200, competitive pricing will doubtless make it a major contender in the home microcomputer market this year. No price was announced at the conference, but the word was that the list price would be well under $1000.



The Atari XL.

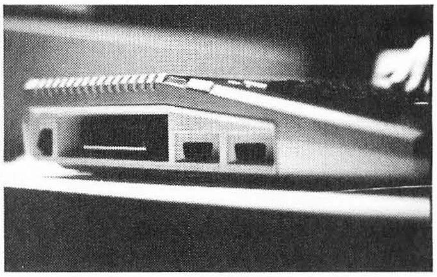
Other features of the unit are the following: keyboard disable function; auto screen shut-off when untended; help key; LED power, keyboard lock, and character set indicators; and one touch cursor movement.

The ROM cartridge slot and controller jacks have been moved to the side of the machine, and number exactly half that of the Atari 800; one cartridge slot and two controller jacks. The determination was made that this was quite enough, and that an extra slot or controller jacks would have only added expense to the machine. There has been no scrimping on the keyboard, however. It is of the highest quality.

New Peripherals, Too

Three new peripherals were announced along with the 1200XL. The 1010 program recorder will allow inexpensive storage and retrieval of data using audio cassettes. The unit features data and audio channels, as did its predecessor, the model 410. It will list for $99.95.

The model 1025 80-column printer will list for $549. It is a customized Okidata Microline 80, and will run in serial at a claimed speed of 40 cps. The dot matrix print is clear and crisp, though not of letter quality.



On the left-hand side of the machine is a single cartridge slot and two controller jacks.

John Anderson is an associate editor for *Creative Computing* magazine.

The unique 40-column color printer/plotter, dubbed the model 1020, will offer text and graphics in four colors at a list price of $299. It will be capable of changing the size and style of its character sets, and 16 colors of pen will be available.

The only disk drives I saw in all my snooping about were the old model 810 clunkers, which are compatible with the 1200, but certainly look out of place next to them. I expect we will be seeing a new drive from Atari within the next half year — conceivably a 3½" model, as compact as the new 1010 program recorder.

New Software Announced

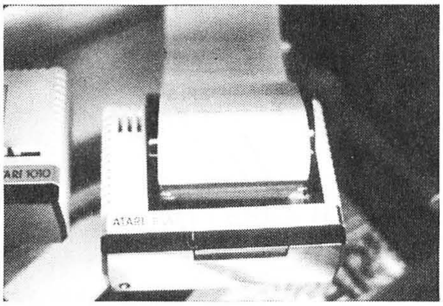
Along with the new hardware, a number of new software packages were announced. *Defender* and *Galaxian* were on hand and running at the Plaza, and should be available now. Both looked to be very high quality clones of their arcade namesakes. I was especially impressed with *Defender;* as was the case with *Pac-Man,* the Atari computer version makes the VCS version look embarrassingly primitive.

E.T. Phone Home will evoke the film *E.T.* with hi-res graphics and fine-scrolling across four screen widths. You are Elliott, helping little E.T. place that long distance call.

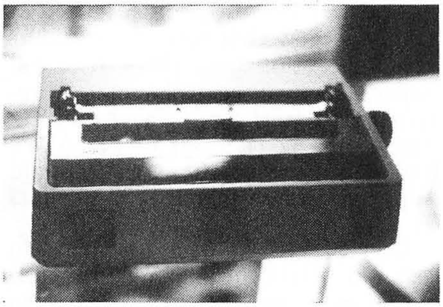
Four other arcade game adaptations have been announced by Atari as well. *Dig Dug* is a popular coin-op maze game, a bit like Pac-Man actually digging his path as he goes.

Qix is a unique and engaging video game. The object is to surround *Qix* with boxes of color. The game transcends the "twitch" aspect with strategy and a lack of patterned play.

Donkey Kong and *Donkey Kong Junior* have also been licensed to Atari, and will become available for the 400/800/1200 soon. These extremely popular coin-op titles will be available within a couple of months.



The Atari 1020 is a 40 column, four color printer and plotter. At a list price of $299, it offers much versatility.



The Atari 1025 80-column printer is the equivalent to the Okidata Microline 80.

Family Finances has been designed to keep detailed records of family income and expenses as well as establish a budget. It is available on diskettes only.

Timewise turns the Atari into an electronic calendar, offering basic time management programs for the home, office, or school. While keeping track of appointments, holidays, and other special dates, the program will also print out schedules and calendars.

Atari Writer is a ROM-based word processor that runs in 16K. It can save files to disk or cassette.

Atari Music I is the first in a series of Music Learning Software. It teaches fundamentals of music theory, including note reading, steps, major scales, and major keys. The four lessons of the program use tutorials, exploratory modes, drills, tests, and built-in video games to reinforce concepts.

Juggle's Rainbow is the first in a series of Early Learning products designed to teach pre-reading skills to children of three to six years. Using graphics and sound, *Juggle's Rainbow* teaches children the concepts of above, below, right and left. Line and circle games help children learn to distinguish between the "tricky" letters, b,d,p, and q.

Juggle's House uses the same techniques as *Juggle's Rainbow* to teach the concepts of upper/lower and inside/outside.

Atari continues to evidence a sensitivity to and understanding of the consumer microcomputer market. There is utterly no doubt that Atari will remain a leading contender in graphics and sound machines for some time to come.

For more information, contact Atari Incorporated, 1264 Borregas Avenue, P.O. Box 427, Sunnyvale, CA 94086.

The Atari 1010 program recorder handles a digital and audio track.

A Letter Quality Alternative For Atari Users

Nancy Blumenstalk Mingus

You want letter quality capabilities on your Atari system, but Atari makes only dot matrix printers. So you decide to wait until they produce a letter quality printer instead of fighting with interfacing to non-Atari printers, right? Well, you don't really have to wait. By using the Atari 850 interface module you can use any parallel printer or RS232 serial printer on the market.

But, be forewarned. Interfacing other products to the Atari is not as easy as Atari would lead you to believe. I found this out the hard way. Although Atari does give you all the information necessary to utilize the interface module properly, you must

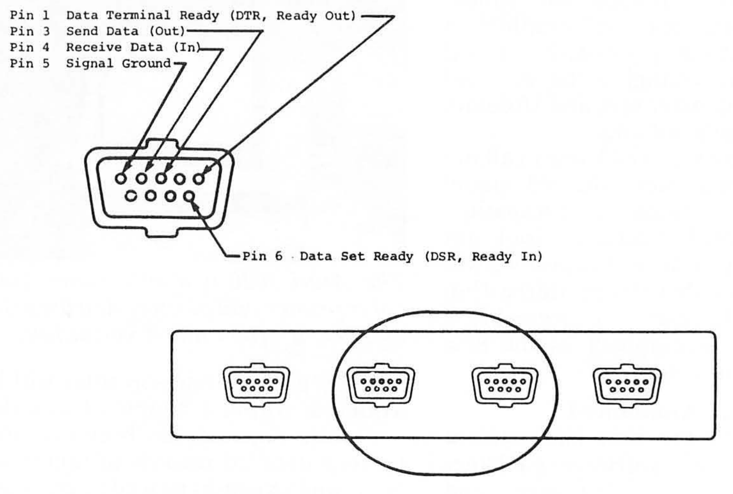
Nancy Blumenstalk Mingus, 15 E. Genesse St., Wellsville, NY 14895.

glean the facts out of three different manuals, and that takes a considerable amount of time. This article will explain some of the problems encountered in interfacing an Anderson Jacobson 832 (RS232) to the Atari, and give some solutions that should apply to many other RS232 printers as well.

One of the hardest problems in using an RS232 printer on the Atari is getting a connector cable between the printer and the interface module. Although the literature on the 850 says it is standard RS232, it only has 9-pin connectors. Most printers use a 25-pin connector. So you need a cable to convert the proper signals coming from the 25-pin connector into signals recognized by the 9-pin connector. Now I'm no electronics expert so I wouldn't even attempt to create my own cable. Since I use port one on the 850 to connect a modem, I wanted port two to be my printer port. With the wiring diagram for port two and the wiring diagram for the Anderson Jacobson printer in hand, I located a good electrician and he kindly wired everything correctly for me. If you don't know anyone in your area who does this kind of work, your local Atari dealer should be able to help you.

Once everything is connected properly, there are a few other things you need to remember when using the printer. The most important of these concerns the disk drive. If you plan on using a disk drive with your interface module and printer you must have DOS II as your disk operating system. There is a special file in DOS II called AUTORUN.SYS which automatically runs when you turn on the computer. This affects the power up sequence you use. You must turn on the printer, then the disk drive, then the interface module and finally the computer. Also, be sure you have the BASIC ROM PAC in place.

Most of the above procedure is explained in the 850 manual, but they make no mention of the AUTORUN.SYS file. The problem we kept encountering was error number 130 when we tried to open the printer port. This message indicates that the interface module can find no such device. What had happened was that the System diskette we had created from the Master diskette had not copied the AUTORUN.SYS file. Once we copied the file on to the System diskette, we could then open the printer port.


Pin 1 Data Terminal Ready (DTR, Ready Out)
Pin 3 Send Data (Out)
Pin 4 Receive Data (In)
Pin 5 Signal Ground
Pin 6 Data Set Ready (DSR, Ready In)

Pin functions of Serial Port Nos. 2 and 3 in 850™ Interface Module (9-pin female connector)

Trying to print or list on the printer is now a fairly simple matter. If you want to print to it you do the following:

OPEN #2,8,0,"R2:"

where:

OPEN signifies initialization of a device or file.

#2 indicates the channel number being used. It can be any number one through eight.

8 means opening the channel for output only, which is all that is required for a printer.

0 this argument is not used, so will always be zero.

"R2:" refers to the port being used. The two signifies port two.

Then any subsequent printing statement would take the following form:

PRINT #2;"Anything",variable

where:

PRINT is the standard BASIC PRINT command.

#2 is the channel previously defined in an OPEN.

Last, to close the device or file, you would enter:

CLOSE #2

where:

CLOSE means you are terminating the use of a channel.

#2 is the channel being closed.

Again, this is explained fairly well in the interface module manual. The LIST command however, is somewhat confusing. To LIST to the printer, instead of LIST#2 as you might expect, you must type:

LIST "R2"

where:

LIST functions the same as usual.

"R2" is the port number you are listing through.

Now that you know how to connect your printer, list a program and print lines to the printer you're almost ready to start. There's one last problem left to deal with. The default configurations of the RS232 ports as shown in the 850 manual need one modification, because the ports do not send a line feed when they send a

Atari to Anderson Jacobson Wiring Chart

Pin No. on AJ Connector	Signal Name	Pin No. on Atari Connector	Signal Name
H	Signal Ground	5	Signal Ground
C	Output	4	Receive
B	Input	3	Send
6	DTR	1	DTR
F	DSR	6	DSR

carriage return. That is, when the print head returns to the left margin of the paper, the paper does not roll up one line. So all those lines you know how to print, print right on top of each other. This produces an interesting effect, yet it is impossible to read. To change this, you must reconfigure this one aspect of the printer port you are working with. Again, I use port number 2 and the command I use is:

```
XIO 38,#2,64,0,"R2:"
```

where:

XIO — is a special command used to configure various aspects of a port.

38 — is the particular XIO command type.

#2 — is the channel number being used.

64 — is a decimal code meaning turn on line feed.

0 — as in the OPEN command, this argument will stay zero.

"R2" — is the printer port.

Before you can do a list you must enter this command, and to be safe, you should include it in any program you write where you plan to print more than one line on the printer.

And now you should be in fairly good shape to start using that nice letter quality printer.

Good luck.

The Atari Word Processors

Phillip Good

Word processing may not be the application of choice for the Atari 800 Home Computer, but there are many times when an Atari user would like to set words on paper in a tidy fashion.

This goal is attainable using any of the five programs described here.,Unfortunately none of them offers all the features of the best of the word processors available for the Apple or the TRS-80.

Let's have a look at what is available, and perhaps you will discover the one that is best for you.

An Unfriendly Keyboard

The looks of the Atari 800 are deceptive. Superficially, the keyboard resembles that of the IBM Selectric, but the right shift key is one silly little centimeter to the right of where an experienced typist expects to find it.

The quotation mark, used constantly in Basic programs, is over the 2 rather than next to the return key. The clear screen key is where the underline should be and is too close to the end parenthesis ")" for comfort.

Corrections are made with a full-screen editor using the cursor almost as if it were a correcting pencil. But the cursor control keys on the Atari are all in shift mode; you must depress the control key each time you use the cursor.

On the other hand, the Atari does offer upper/lower case capability without hardware assists, and, unlike the Apple, the shift key is fully functional as delivered.

Like the Apple, the Atari 800 offers only a 40-column display. Unlike the Apple, no one has yet marketed an 80-column adapter. And none of the three Atari full-screen word processors makes use of the Atari high-resolution graphics to generate a 60+ column character set.

Bare-Bones Word Processors

For less than $20, either of two bare-bones word processors will allow you to use the Atari to create and edit messages for an electronic bulletin board, display them on the monitor or TV screen, store and retrieve text and programs, and produce a hard copy.

With *Letter Writer*, $19 from CE Software, you use the insert key to insert text, and the delete key to delete text errors. The program provides only two editing features of its own to let you indent paragraphs and skip lines.

The *Letter Writer* printer formatter allows you to set the line length (though not the left margin), insert new pages as required, and right justify your text. The program will operate with any parallel connected printer. I used *Letter Writer*, a $30 interface cable from Mactronics Inc., and a C. Itoh printer to prepare some reports recently.

But *Letter Writer* is still not a best buy. That honor goes to *Bob's Mini Word Processor*, which costs just $15 from Santa Cruz Educational Software.

The *Mini-Processor* allows you to create files, save or load them, modify them, and create hard copy. While editing, you have full control over the tab, delete, back space, clear, insert, and cursor control keys. But you can also advance through the text a page at a time or move with a single command to the beginning or end of the text. You may interchange "pages" of text, though you cannot cut and paste any section smaller than a page.

The *Mini-Processor* works with serial but not parallel printers, using an Atari 850 interface.

Inadequate Documentation

Ideally, any software package should include four types of documentation:

- Tutorials to get you started; the more examples and the more demonstration files, the better.
- A quick-reference manual including a detachable command reference card to keep near the keyboard.
- A comprehensive reference manual.
- Application notes for programmers.

Figure 1.

Method One: Menus

A block of text can be deleted using the menu tree.

1. Repeat steps 1-4 under "Delete Next Character".
2. Recall the page that contains the text to be deleted.
3. Type E in response to the next menu prompt.
4. Press return.
5. Type T in response to the next menu prompt.
6. Press return.
7. Type S in response to the next question.
8. Press return.
9. Position cursor at the beginning of the block to be deleted.
10. Type G and press return. A right parenthesis in inverse video will appear in column one and a blank line will be inserted.
11. Position cursor after the last line of text to be deleted.
12. Type D in response to the next question.

Philip Good, Information Research, 10367 Paw Paw Lake Dr., Mattawan, MI 49071.

Atari's *Word Processor* has all four, but the tutorials and reference manual are completely incomprehensible. The combined manuals are more cumbersome (and bulky) than any of the more than sixty manuals I recently reviewed. Figure 1 shows instructions for deleting a block of text from pages 31-32 of the Atari manual.

See, it's as easy as a,b,c,d,e,f,g,h,i,j,k,l. Don't ask how to move a block of text; that takes 28 steps (Method 1). Whoever wrote this Atari manual (I think it was a committee) also wrote the mainframe manuals that drove us to using personal computers in the first place.

All three word processors—Atari's, *Letter Perfect*, and *Text Wizard*—do provide detachable quick-reference cards. *Text Wizard* has only one example, which you must type in yourself, and no demonstration files. The *Text Wizard* tutorials also serve as the comprehensive reference section—or is it vice versa? My manual was missing a page—the page that told how to save the file I had just created.

Letter Perfect has no demonstration files and only one example—a form letter. You must buy a second LJK product for another $150 to make use of the example.

Text-Editors

The Atari *Word Processor* has the best text-editor of the three full-screen word processors, if you can figure out how to use it. You can display text on the screen as it will appear in print. You can work with files much, much larger than memory. And you will automatically save what you have edited as you move from page to page. (Unfortunately, you will destroy the old text as you do so; back-up is not automatic.)

Text Wizard is the only one of the three which lets you edit programs as well as text, enter insert mode for the rapid insertion of many paragraphs of text, and move or copy entire blocks of text simply and rapidly.

Letter Perfect is the only one of the three to provide a block delete safeguard, and to let the user set tabs with a cursor.

There is an extensive list of simple editing functions that can't be done with any of the three including:

- Print one file while editing another.
- Display a second file.
- Automatically back-up on file-save.
- Insert key phrases with a single keystroke.
- Use wild cards in a search and replace command.

Printer-Formatters

You will probably have to settle for less than letter quality with an Atari. None of

Table 1. Atari Text Editor.

	Atari WP $150	**Letter Perfect** $140	**Text Wizard** $99
Overall			
years on market	1/2	1/2	1/2
back-up	no	$20 by mail	$5 by mail
uses Hi-Res graphics	no	no	no
menu driven	yes	yes	no
can display multiple files	no	no	no
displays text on screen as it will appear in print	yes	no	no
can print one file while editing another	no	no	no
handles files larger than memory	yes	no	no
can edit programs as well as text	no	no	yes
control characters can be customized	no	no	no
Documentation			
getting started	slow	slow	easy
tutorials	hopeless	no	no
examples	yes	none	one
help menus	cumbersome	no	no
reference material	cumbersome	yes	good
separate reference card	yes	yes	yes
File Control			
continuous back-up	by page	no	no
save file and continue editing	yes	yes	yes
automatic back-up on file save	no	no	no
file protect safeguard	yes	yes	no
insert a second file with one command	yes	yes	yes
insert a portion of a second file	no	no	no
display a second file	no	no	no
display file directory	no	yes	yes
kill file (and create space)	no	yes	yes
can prepare files for transmission	no	no	yes
Scrolling (or cursor movement)			
by word	no	no	no
by line	yes	yes	yes
by sentence	no	no	no
by screen	yes	no	yes
to beginning or end of workspace	yes	yes	yes
to beginning or end of document	yes	no	no
horizontal scroll	yes	no	no
Delete			
by word	yes	no	no
by line	yes	yes	yes
by sentence	no	no	no
delete recover	yes	no	no
by screen	yes	no	no
by block	yes	yes	yes
block delete safeguard	no	yes	no
continuous delete	no	yes	no
Insert			
keyphrases	no	no	no
typeover (fast)	yes	yes	yes
insert mode (for many words)	no	no	yes
push ahead (for one or two letters)	yes	yes	yes
split and glue a line at a time	no	no	no
intermediate buffer	yes	yes	no
block whole sections	complex	no copy	yes
delete and restore	yes	yes	no

The tables shown are reproduced from "Choosing a Word Processor," by Phillip Good. Copies of this book may be obtained from Information Research, 10367 Paw Paw Lake Drive, Mattawan, MI 49071. Cost is $14.95 plus $2.00 for shipping and handling.

the full-screen Atari word processors reviewed here supports the special features of a Qume or a Diablo. Atari owners must content themselves with one of two dot matrix printers—the Atari 825 (the Centronics 737 in disguise) or an Epson MX-80. The Epson is by far the better buy, even though it will not support underlining, superscripts, or subscripts.

You can't alter the number of lines per inch with any of the three full-screen word processors. You are limited to a one-line heading. You can't use soft or phantom hyphens; that means you will need to spend time printing and reformatting until you get it right. You will spend less time with the Atari word processor perhaps, because it lets you view your material on the screen just as it will appear on the printer. But the screen display is so inefficient and time-consuming, you may find it faster to use the print and guess method of *Text Wizard* or *Letter Perfect*.

None of the three lets you interrupt and resume printing, whether to answer the telephone or to pause for text entry from the keyboard. None of the three has mail-merge capability. You can get mail-merge capability for *Letter Perfect* by purchasing LJK's *Data Manager*. A mail merge option for *Text Wizard* is in the works.

A Lost Cause?

I don't think the Atari is a lost cause. With very little programming effort, one can correct its keyboard deficiencies. The cursor control keys can be reprogrammed for lower case use. This has already been done by Eastern House Software in their *Atari Monkey Wrench*. The Atari high resolution graphics can be used to create a 60+ column display without hardware assists. Both of the bare-bones word processors already support letter quality printers; there is no reason the more expensive full-screen word processors cannot provide the same support. □

Vendor's List

Bare-Bones

Bob's Mini-Word ($15), Santa Cruz Educational Software, 155425 Jigger Dr., Soquel, CA 95073. (408)476-4901.

Letter Writer ($20), CE Software, 238 Exchange St., Chicopee, MA 01013. (413)592-4761.

Full-Screen

Letter Perfect ($140), LJK Enterprises, P.O. Box 10827, St. Louis, MO 63129.

Text Wizard ($99), DataSoft Inc., 19519 Business Center Dr., Northridge, CA 91324. (800)423-5916.

Word Processor ($150), Atari Inc., 1265 Borregas, Sunnyvale, CA 94086. (800)538-8543.

Table 1. continued

Search	**Atari WP**	**Letter Perfect**	**Text Wizard**
find phrase anywhere in document	yes	*	*
find with user option to replace	yes	yes	yes
find and replace n times	no	no	no
find and replace all in document	yes	no	no
find and replace all in memory	yes	no	no
use wild cards	no	no	no
ignore upper/lower case in matching	no	no	no
Screen Format			
format entire text	yes	no	no
format different parts differently	no	no	no
set line length	yes	no	no
set tabs with cursor	no	yes	no
set tabs by command	yes	no	no

* Not applicable.

Table 2. Atari Text Formatter.

	Atari WP	**Letter Perfect**	**Text Wizard**
Overall			
display on screen as it will print	yes	no	no
print one file while editing another	no	no	no
mail-merge or file-merge	no	extra $	extra $
letter quality printers supported	no	no	no
Layout			
set from a menu	yes	no	no
menu may be skipped	yes	---	---
under user control while printing	no	no	no
characters per inch	yes	yes	yes
lines per inch	no	no	no
width limitation	80	80	80
Page Control			
one line heading	yes	yes	yes
multi-line heading	no	no	no
heading and footing	no	yes	yes
page numbering	yes	yes	yes
odd/even page distinction	yes	no	no
conditional new page	yes	no	no
Text Control			
justify	yes	yes	yes
center	yes	yes	yes
phantom hyphen	no	no	no
conditional formats	no	no	no
multiple columns	yes	no	yes
reverse line feed	no	no	no
Printer Control			
underline	yes	yes	yes
bold face	yes	yes	yes
vary bold face intensity	no	no	no
super- and sub-script	yes	yes	yes
change ribbon colors	no	no	no
kerning	no	no	yes
change control characters	no	no	no
proportional spacing	yes	yes	yes
Output Control			
interrupt/resume	no	no	no
pause for text entry from keyboard	no	no	no
pause for variable entry	no	no	no
start/stop at designated page/record	yes	yes	yes
print multiple documents	no	no	yes
print multiple copies	no	yes	no

Atari Text Editor Program

Elwood J. C. Kureth

I'd be willing to bet that a fair number of people who own a computer and a line printer do not own a typewriter. Of those individuals who don't own one, it would probably be safe to assume that its absence could be attributed to the fact that (a) the need for a typewriter has never arisen, or (b) they don't need (or use) a typewriter often enough to warrant purchasing their own.

If you already own a typewriter (as I do) in addition to your computer and its related items, and you already type with confidence, then perhaps this program will be of little use. However, if you're like me, you usually end up making a few mistakes, which means erasing or starting all over again.

This program was written for an Atari 800 with an Epson MX80F/T printer. It's not a word processing program by any means; in fact, it's very limited in its application. What it allows you to do however, is put text on the screen, edit it, and send it to a printer in two different print modes.

RUN it and you will be asked to set the right margin (up to 80 columns). Hit a RETURN to enter the number. You will then have to determine if you want emphasized print. This type of print is much bolder than normal print and approaches letter quality. Simply type "Y" or "N" (no RETURN is necessary because the keyboard "reads" your input). Next, you will be prompted for single or double space. After your selection, you will face a blank screen. The first key you hit will display the cursor, and away you go.

Four spaces from the end of each line a warning buzzer will sound, just like the bell on a typewriter. The cursor will not advance once you have reached the right margin; it will, however, backspace or RETURN. So there's no need to worry about overrunning your margin.

NOTE: A HEART (CHR$(O)) will appear each time a RETURN is hit. The heart will help you keep track of your lines on a 40 column screen.

Let's say you have a 37 or less character line on the screen (79 or less character line for an 80 column screen), and you want to change a character. If the cursor has already advanced down one physical line (due to a RETURN or end-of screen return) you will be unable to correctly edit the line the cursor just left. If you wish to make a change to a line of text, it must be done while the cursor is on that line.

You may move the cursor backwards by using either the DELETE/BACK S key or the CTRL<--keys. Using the DELETE/BACK S key will delete the character the cursor covers. Let's say you have the word "MICROOCOMPUTERS", and you wish to delete the second "O" from that word. This action could be accomplished one of two ways. The DELETE/BACK S key could be used until the cursor is over the second "O", thereby deleting it, as well as all the characters that had followed it. Now it would be necessary to retype the rest of the word.

The alternate method would be to use the CTRL<--keys, moving the cursor backwards to the *immediate right* of the second "O" (cursor would be covering the "C"). At this point you would hit the DELETE/BACK S key, which would move the cursor over the "O" (deleting it), followed *immediately* by the CTRL-->keys to the point where you'd left off. The cursor will automatically stop at that point if you hold the keys down.

CAUTION: With the exception of the abovementioned example, anytime you move the cursor backwards, your *first* action when moving it forward again must be to type at least one character, as opposed to immediately using the CTRL-->keys to start moving the cursor forward. If, in the example used above, you overshoot the "O" and the cursor winds up over the "M", instead of using the CTRL-->keys to move the cursor, you would first type the letter "M", then CTRL-->(or type) to the right of the "O", then DELETE/BACK S over the "O", then (whew!!) CTRL-->to the point at which you started backwards.

The last line of text must be followed by a RETURN. Then, it's simply a matter of hitting a CTRL P, and your text is transfered from screen to paper. More copies? Just hit a CTRL P.

If you desire to type new material, you must first clear the memory by hitting the ESC key. If this is not done, two things can happen when you print out the new text. First, the previous material will be printed out before the new text. Secondly, if you hit a SYSTEM RESET, run the program, type new material, *and edit* that new material, you could get a rather confusing text. Always hit the ESC key *first* after you're through printing your material.

```
1 REM ATARI VERSION--BY ELWOOD J.C. KURETH,JR.
5 OPEN #1,4,0,"K":OPEN #7,8,0,"P":GRAPHICS 0
10 POKE 752,1:M=0:COUNT=0:BUZZ=0:? #7;CHR$(27);CHR$(64)
15 ? CHR$(125):POSITION 5,5:? "SET RIGHT MARGIN (UP TO 80)";:INPUT MARGIN:IF MAR
GIN>80 THEN GOTO 15
20 POSITION 2,5:? "DO YOU WANT EMPHASIZED PRINT(Y OR N)":GET #1,LTTR:IF LTTR<>89
 AND LTTR<>78 THEN GOTO 20
22 ? CHR$(125):POSITION 5,5:? "SINGLE OR DOUBLE SPACE(S OR D)"
23 GET #1,SPACE:IF SPACE<>83 AND SPACE<>68 THEN GOTO 22
24 ? CHR$(125);:POKE 752,0
25 GET #1,IT
26 IF IT=156 OR IT=157 OR IT=254 OR IT=255 OR IT=125 THEN GOTO 25
28 IF IT=126 OR IT=30 THEN GOTO 500:REM BACKSPACE
30 IF BUZZ=MARGIN AND IT<>155 THEN GOTO 25
35 IF IT=16 THEN GOTO 600:REM PRINT
37 IF IT=31 THEN GOTO 900:REM ADVANCE CURSOR
38 IF IT=27 THEN GOTO 2000:REM CLEAR MEMORY
40 COUNT=COUNT+1:BUZZ=BUZZ+1:IF BUZZ=MARGIN-4 THEN ? "}";
41 IF IT=155 THEN ? CHR$(0);:GOTO 1000:REM RETURN
42 M=M+1
45 POKE 6000+M,IT:? CHR$(PEEK(6000+M));:GOTO 25
500 IF IT=126 THEN POKE 6000+M,0
510 ? CHR$(IT);:GET #1,IT
515 IF IT=126 OR IT=30 THEN M=M-1:BUZZ=BUZZ-1:GOTO 500
520 IF IT=155 THEN GOTO 1000
530 GOTO 45
600 FOR X=1 TO COUNT:IF LTTR=89 THEN ? #7;CHR$(27);CHR$(69);
620 ? #7;CHR$(PEEK(6000+X));:NEXT X:GOTO 25
900 M=M+1:BUZZ=BUZZ+1
905 GET #1,IT
907 IF M>COUNT AND IT=31 THEN GOTO 905
908 IF IT=31 THEN ? CHR$(IT);:GOTO 900
910 IF IT=155 THEN GOTO 1000
920 GOTO 45
1000 COUNT=COUNT+1:POKE 6000+COUNT,155:? CHR$(PEEK(6000+COUNT));
1010 IF SPACE=68 THEN COUNT=COUNT+1:POKE 6000+COUNT,155:? CHR$(PEEK(6000+COUNT))
;
1020 M=COUNT:BUZZ=0:GOTO 25
2000 POKE 752,1:? "}":POSITION 7,5:? "****  PLEASE WAIT ****"
2010 FOR FILL=1 TO COUNT:POKE 6000+FILL,0:NEXT FILL:? CHR$(125):GOTO 10
```

Elwood J.C. Kureth, HHD, 14th Maintenance Bn., APO New York 09169.

VisiCalc: Reason Enough for Owning a Computer

Doug Green

Ideally your computer should be able to act like a cross between an electronic piece of paper and a pocket calculator. That seems to be just what the people at Personal Software, Inc. had in mind when they developed VisiCalc. VisiCalc is not merely a piece of interactive software, but in some respects is more like a separate programming language. It is extremely powerful, and handles many varied jobs with aplomb. When used properly it can save a great deal of time that would ordinarily be spent programming or using several pieces of software. VisiCalc cannot do some of the things that high level languages can do, but what it can do, it does very well indeed.

It takes much less time to learn virtually everything there is to know about the VisiCalc system than it takes for any other programming language you can think of. In my case it took about seven days averaging about one and one-half hours a day to become conversant with all that VisiCalc has to offer. This is in sharp contrast to the various high level programming languages that demand much more of the learner in exchange for their greater flexibility.

Not only does it take only a short period of time to understand the entire VisiCalc system, but it takes almost no time to begin getting results from this remarkable piece of software. This is an opinion that I share with everyone that I have demonstrated this system to, as well as several people in the computer business who already use VisiCalc or supply it to other users.

A Window Into The Computer's Memory

After you load in the VisiCalc disk you will have the basic electronic sheet of paper on your screen. As you can see from Photo 1, it has 20 rows and four columns. Each location in this grid is identified by the number of the row and the letter-code at the top of the column, for example, A1. The cursor in VisiCalc is much wider than the usual single-character cursor; it takes up the entire entry that it occupies on the grid.

Any entry on the sheet can either be a number, a word, or a function of the contents of other locations. This is one of the reasons that VisiCalc is so powerful. Whenever a location is changed by the user, *all of the locations that depend on it are automatically recalculated.* It is this aspect of VisiCalc that is so striking and so useful.

Let us say you have told the VisiCalc sheet to derive column C in some way from columns A and B. Then if, for some reason, you change any of the values in columns A or B, new results in column C will be displayed automatically. This is like using FOR . . . NEXT commands in immediate mode without ever having the contents of your memory leave the screen.

Although what you see is limited by the number of spaces that can be displayed on your screen at once, the electronic sheet is actually much larger. There are 254 rows and 63 columns where information can be stored, and the amount you can store is limited more by the size of your computer's memory than it is by the VisiCalc sheet.

Keeping track of the remaining memory is very simple since it is constantly displayed in the upper right hand corner of the screen.

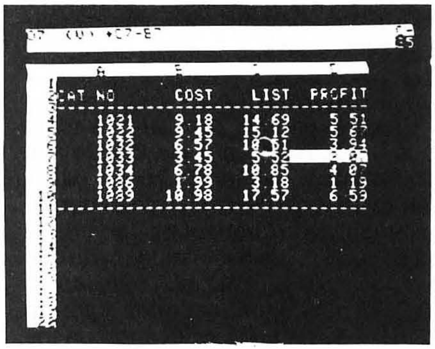
You may only see 20 rows of data at one time, but the number of columns can be varied by changing the width of the columns. You can also store more information in one of the grid locations than it appears able to hold. The system will remember exactly what was entered regardless of how narrow you choose to make the visible columns. The screen will display as many characters as you allow for, beginning from the left of your input.

In addition to the grid, there is space at the top of the screen where other important information is displayed.

The white bar displays the contents of the location where the cursor is currently residing. This can either be a value (v) or a label (l). These terms are analogous to numeric and alphanumeric variables that one deals with when using Basic; except just a value can be an expression referring other locations in the table.

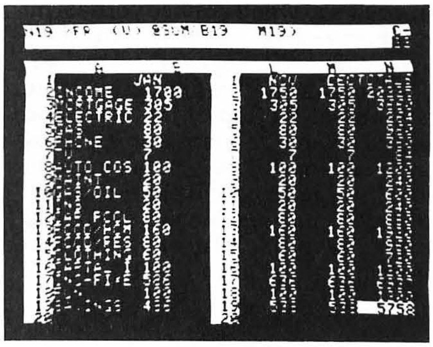
Two Independently Scrollable Windows

If you are not satisfied with the information that you can see on the screen at one time, you can split the screen in either the horizontal or vertical direction and look at whatever portion of the sheet you like in either window. A common use of this feature is to display the upper left corner of your sheet in the left window while the lower right portion of your work is displayed in the right window. That way you can change your initial entries and watch your totals change at the same time. Photo 2 shows an example of how this might be put to use while analyzing the family budget for the upcoming year. Instead of wondering idly what would happen to your savings for the year if the electric bill goes up five dollars a month, you can find out just by typing over the information that you would like to see changed. As you might guess, this will change the entire row that lies beyond the changed data, along with all of the column totals that depend on these figures.



```
D7  (V) +C7-B7
CAT NO      COST    LIST  PROFIT
--------------------------------
  1021      9.18   14.69    5.51
  1022      9.45   15.12    5.67
  1032      6.57   10.51    3.94
  1033      3.45    5.52  [illegible]
  1034      6.78   10.85    4.07
  1036      1.99    3.18    1.19
  1039     10.98   17.57    6.59
--------------------------------
```

Photo 1.



```
N19 (FR) (V) @SUM(B19...M19)
             JAN
INCOME      1700
MORTGAGE     305
ELECTRIC      22
AUTO_COS     100
HEAT/OIL
                                   5750
```

Photo 2.

The Replication Feature

Another impressive feature of this system is the ability to replicate similar functions down a row, across a column, or in both directions at once. For example, if you wish to have VisiCalc derive values for column C by subtracting those in column B from the corresponding values in column A, all you need do is type in the directions for the first location in column C along with directions for replication. This will cause column C to be completed in an instant.

If you are trying to complete a table of entries that depends on the values stored in the top row and the left hand column, all you need do is supply the directions for the entry located at row two, column two along with the replication commands and the screen will fill before your eyes, much faster than most users could

Doug Green, Cortland Jr.-Sr. High School, Valley View Drive, Cortland, NY 13045.

type in the specific formulas to perform such a task.

Cursor Control

The ← and → keys are used to move the cursor from side to side and up and down, while the space bar is used to change the direction of cursor movement from horizontal to vertical and back. For rapid movement you can hold down the repeat key. There is also a GOTO command that allows you to move the cursor to any location on the sheet with just a few keystrokes.

The little dash in the upper right hand corner of the sheet tells you which way the cursor is currently prepared to move. The letter next to this dash, either a C or an R, lets you know the current direction that the recalculation will occur in. You can instruct VisiCalc to recalculate down the columns (C) or across the rows (R). This will depend on how you have set up the entries in your table.

The ESC key is used to recover from simple typing mistakes. If you press it often enough it will erase all that you have typed in since you last hit the return key. As you enter data for a given location it appears on the so-called prompt line, the line between the white box at the top of the sheet and the grid. When you close an entry by hitting return, or moving the cursor to another location on the page, the contents of the prompt line are calculated (if necessary) and placed in the location on the grid that you have just dealt with.

More Functions And Commands

There are a number of other functions that are available to VisiCalc users. These are all listed in Table 1, but a few deserve special mention. The sum function is especially useful to anyone dealing with columns of numbers that must be added. (Think of all the time operators of small businesses can save by not having to bang number after number into a calculator. With VisiCalc they only need to be written once.) You can also ask for the average of a range of values along with other common functions used in business, science, and mathematics.

The list of commands is also impressive. With a few key strokes you can blank out any location, add or delete a row or column, move a row or column to a new location on the page, or repeat a number or letter across any location in the grid. This last command is especially useful for drawing lines across the page like those in Photo 1. There are a number of commands that can change the format of a given location or the entire window that the cursor is located in. The choices for these format commands include: general, integer, dollars and cents, left- or right-justified columns, and graphing. This final command can be used to construct simple bargraphs for information displayed in a range of entries selected by the user. This is shown in Photo 3.

Other commands couple or uncouple the movements of pairs of windows, fix the titles on the screen

VisiCalc Functions

SUM	Calculates the sum of the values in a list
MIN	Calculates the minimum value in a list
MAX	Calculates the maximum value in a list
COUNT	Results in the number of non-blank entries in a list
AVE	Calculates the average of the non-blank values in a list. The maximum number of values in the list is 255.
NPV	Calculates the net present value of the cash flows in a list, discounted at the rate specified. The first entry in the list is the cash flow at the end of the first period, the second entry is the cash flow at the end of the second period, etc.
LOOKUP	Used with a list of items that are ranked in ascending order. This function returns the value from the list that is less than or equal to the value referenced in the command given.
PI	Returns the value of 3.1415926536
ABS	Returns the absolute value of the value given
INT	Returns the integer portion of the value given
EXP SQRT LN LOG10 SIN ASIN COS ACOS TAN ATAN	Calculates the appropriate function. The trigonometric calculations are done in radians
NA	Results of a calculation are not available. This makes all expressions using the value display as NA.
ERROR	Results in an "Error" value that makes all expressions using the value display as ERROR.
>>>>	This means that there is not enough room to display the calculated value in the room available. Making the columns wider will often allow the value to be displayed.
Scientific Notation	VisiCalc will automatically shift to scientific notation if necessary in order to display a value in the space alotted.

Table 1.

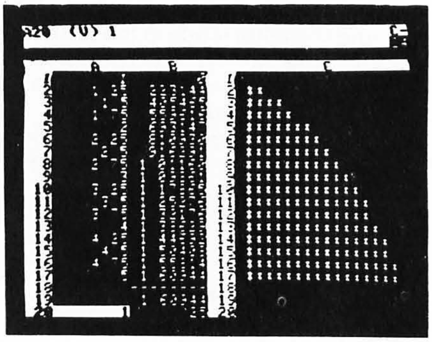


Photo 3.

as the cursor moves down or to the right, and replicate formatting across a whole column or row, or the entire contents of the current window. These commands require between two and five keystrokes each depending on what is being accomplished. (The Clear command requires three keystrokes, a fact that saved me from clearing the VisiCalc sheet at a time when I was really trying to do something else.)

VisiCalc manages its own storage in its own format. It provides storage commands allowing you to save files on disks or cassette tapes, load files from a disk or a cassette, delete a file from a disk, or initialize a blank disk so that it will be ready to receive VisiCalc files for storage. It is easy to ask for a list of the file names on a given disk. You can also print the contents of your sheet on a disk as a "text file." This file can be read by other programs in Basic, for example, and the information can be further processed in this manner. (This feature permits you to perform whatever other functions you may feel are missing.)

Similar commands will result in the printing of your electronic sheet by your printer. The output will be what is actually on the sheet, as opposed to what appears in the window, so be sure to pay attention to the line width of your printer. In any case you can specify the portion of the page that will be printed with the issuance of the proper print command.

Stay Tuned

Your purchase of the VisiCalc package includes an instruction book that contains an introduction and four lessons. As I read through the book and carried out the examples I found the text to be easy to understand. The explanations were certainly cleaner and better than those I have seen in most systems programing manuals. Along with the book, which is in a handsome 10 x 7½ inch three-ring binder, you receive the VisiCalc reference card. This contains a summary of all of the VisiCalc commands and functions and is extremely useful for users who are new to the system. It would also be invaluable to infrequent users. When you send in your warranty card you will receive the first copy of the VisiCalc Newsletter free. Original owners are also protected from any defect in the disk for 90 days, and replacement thereafter for $15.00.

The people at Personal Software, Inc. are planning to improve the system and offer the updated versions to original owners at a reduced price. They also encourage users to suggest changes and additions to improve the system. As a VisiCalc user I would suggest that they add some of the more commonly used statistical functions to those listed on Table 1. The ones that I would suggest would be: standard deviation, one or more correlation coefficients, and perhaps the ability to do a t-test and a least-squares linear regression; but new functions, must use up too much memory.

Machines And Memory Requirements

Although the version I used was designed for an Apple system, it will soon be available for other makes of small computers including Pet and Atari. It is only available on disk and requires a minimum of 32K of RAM. Additional memory will allow for the storage of a much larger electronic sheet but all of the systems' features are available for users of 32K systems.

The version that I used (version 35) requires 23K for the resident program. This means that for a 32K system there remains only 9K for storage of the electronic sheet. This still allows for a reasonable amount of storage, but for most business applications it would be a good idea to have 48K available.

Worth The Money?

If you are in business, the chances are that the cost of a VisiCalc disk will be one business expense you will gladly bear. The current suggested retail price is $150.00. This may be a bit steep for someone who only needs to do his check book and the family budget, but for almost anyone in business, education, or any science-related field it is not only worth the initial expense, but reason enough to purchase a small computer system in the first place. □

Atari Resources

Where do you get more help and information about your Atari? Obviously, *Creative Computing* is one source, and there are several others. If you are a beginner, the Atari Basic self-teaching guide that came with your computer will get you started. When you send in your warranty card, you will recieve the Atari 400/800 Basic Reference Manual, which is much better, and actually answers most of your questions. I had three questions when I first started programming the Atari:

1. How do you concatenate strings?
2. How do you array strings?
3. How do you obtain keyboard input without stopping the program?

Atari had given me the name of someone in the plant to call for questions, so I called and left my questions. Within hours they called back with the answer; "We don't know." The next day my preliminary reference manual arrived, and it had answers to all three questions! The answers were not easy to find, but they were there.

1. To concatenate a string variable, follow these steps:
   - **a. Dimension the recieving string large enough to hold the combination.**
   - **b. Determine the length of the original string with the LEN function.**
   - **c. Assign the string to be combined to the next location in the receiving string.**

Here is a program to do it:

```
10 DIM A$(10):DIM B$(5)
20 A$="THIS"
30 B$="+THAT"
40 A$(LEN(A$)+1)=B$
50 PRINT A$
```

2. String arrays are difficult in Atari Basic. Essentially, you have to dimension a very large string, store all other string data as substrings, and do your own bookkeeping to keep track of where each item is. The Alpanumeric Sort routine in Appendix A of the Reference manual uses this method. One advantage of Atari Basic is that there is no arbitrary limit to the size of a string, as there is in Microsoft Basic, so there is a lot of flexibility.
3. To strobe the keyboard, PEEK location 764 in memory to determine when a key is pressed. To obtain a single character from the keyboard, OPEN the keyboard as an input device and use the GET command:

```
10 X=PEEK(764):IF X<255 THEN PRINT X
20 GOTO 10
10 OPEN #1, 4, 0, "K:"
20 GET #1, A
30 PRINT CHR$(A)
```

Other sources of information include **Compute!** magazine which divides its attention between the Pet, the Atari, and the Apple. The cost is $20.00 a year for 12 issues. For a subscription, write: **Compute!**, P.O. Box 5406, Greensboro, NC 27403.

Two magazines published soley for Atari owners are **A.N.A.L.O.G.** (6 issues, $12.00 a year) and **ANTIC** (6 issues, $15.00 a year). Write to **A.N.A.L.O.G.** at P.O. Box 615, Holmes, PA 19043 and to **ANTIC** at 297 Missouri Street, San Francisco, CA 94107.

I have since received the regular Basic Reference Manual, and it is even better than the preliminary one. One nice new feature is an excellent memory map. Some information is still not released, but I get the impression that this is because Atari is reluctant to release it in its preliminary form, not because they are trying to hide something. I know that they have been particularly helpful to friends of mine who have signed non-disclosure forms.

Tutorial Series

One excellent source of information is *Iridis*. *Iridis* was first advertised as a magazine, but now describes itself as "a series of tutorials about the Atari Personal Computer." It is sold, not by subscription, but by individual issues.

Iridis I contains four programs with explanatory articles, three columns, and an explanation of their format for printing control characters. You can purchase it either with the programs on cassette ($9.95) or on disk ($12.95).

The four programs include "Clock," a high resolution wall clock with moving hands, ticking and chimes; "Zap," where a joystick-controlled snake moves around the screen eating bits of food and growing; "Logo," which displays the *Iridis* logo in dozens of different shades, with instantaneous changes from one color to another; and "Polygons," which constructs geometric patterns.

Each program is listed, and a "behind the scenes" article following each listing explains the program in detail. These listings are very well done, and contain fascinating glimpses into programming techniques. For example, you can test to see if the START button is pressed by checking to see if memory location 53279 contains anything other than 7.

The three columns are "Novice Notes," with programming tips for the beginner, "Hacker's Delight," which goes into detail about how the machine works, and "Oddments," which contains features too short to deserve an article, but too significant to be ignored.

Iridis I comes in manual format, 6 inches by 9 1/2 inches, and contains 32 pages. The print is quite small, and appears to be typeset with a small computer word processor and printer. Except for a chart on the last page showing the Atari control characters, there are no illustrations.

You may order *Iridis* from The Code Works, Box 550, Goleta, CA 93017

Itty Bits

As a closing feature, here is a calculator program I use frequently to balance my checkbook, do my taxes, and for any other adding machine functions. Although it is very short, it is one of my favorite programs. To clear the memory, enter the present value of the accumulator (B) as a negative number.

```
10 INPUT A : B=B+A : PRINT B : GOTO10
```

Questions and Answers

The following are answers to some of the mail we have received of late.

Q. PEEK(741)+256*PEEK(742) (from July '81) is not a good way to find the display list. PEEK(560) is. Why didn't you?

A: Knowledge about the Atari is a rapidly unfolding thing. We pass on what we know when we know it. And remember, we write columns about four months before you read them. Since we are experimenting with the Atari all the time, and learning more, sometimes we discover a better way of doing things about which we have already written. No matter; we try to give the best of what we know at the time.

Q: In the DLI article (December 1981) you don't use memory page 6. Why? If you did, you could fix the location of the program and avoid the relocation code.

A: First, we left the page alone so the user could use it along with the DLI routine. Remember, the DLI routine will coexist and coexecute along with many assembly routines as it is an interrupt handler. Hence, it is potentially more useful located outside of page 6.

Second, it gives us a chance to explain all about string handling and the general principles behind regarding a string as just a collection of bytes in memory, useful in other ways besides merely holding characters. These are tutorials, remember, and often the stated goal is far less important than the getting there. The principles behind the demonstration will be far more useful, in many ways, than will be the demontration.

Q: In the July article you show a mixed mode display, which I can't produce. Could you send me the code for this? (Multiply this by 80 letters or so.)

We omitted the code because I was addressing the principle of stacking display blocks, and the code is somewhat confusing. It tends to raise more questions that it answers, but I have included it here for the curious. See Listing 1.

Basically we are modifying a graphics 8 display list to:

GR.2
GR.2
GR.0
Gr.8 x lots

We are not duplicating the July display exactly, but you can with the principles in the code.

We use two GR.2 lines to make the memory requirements come out to 40 bytes, to keep "in sync" with graphics 8. We then put data into the first 120 bytes of DM for character output.

Character data is translated from ATASCII to INTERNAL format for display; they are not the same. A machine language routine here would be quite nice; there is probably one in the operating system that could be used. The INTERNAL codes are then POKEd into memory.

Because we now generate 16+16+8+189 scan lines, instead of 192, we have a total of 229 generated lines. This will probably cause your TV to "roll." So we chop out the lower 40 graphics 8 instructions by moving the JVB instruction up. I copy the data bytes first, then the JVB byte, to prevent the JVB taking off into random memory.

Or so we thought. (And so we told you.) JVB is the jump and wait for vertical blank; it makes the display list into a GOTO loop, so we said. Except that just by accident we found out that where it jumps to doesn't matter. That's right: the data bytes following the JVB are irrelevant. Why? Because at the start of every screen refresh, the operating system copies the display list location shadows (560, 561) into Antic and re-sets him to the start of the DL. So all is well even if Antic, at the end of the DL, jumps off to kingdom come.

Except: during disk accesses, where apparently the Vblank routine copy is nulled. Then the screen will go wild. (See what I mean about "rapidly changing knowledge"?)

Along these lines, a fun display is to set up two display lists and two display memories, and have Antic execute them alternately. (Use a DLI in the first 112 instruction to swap display memories.) You'll get two displays superimposed on each other. For example, we had a graphics 0 display of Basic code imposed on the graphics 8 display it produces. Nice, and nifty for an editor or such. However, it does tend to flicker.

Q: Speaking of flickers, your DLI routine has an annoying flicker in midscreen—a border between two colors that jumps back and forth. Why?

A: You're right. Next question?

Seriously, the reason for this is that the 6502 just doesn't have enough time to copy all the data into the CTIA color registers before the TV scan line begins. In fact, it can't even start until midway through the last scan line of the display block with the interrupt flagged. The TV refresh process outruns this rather generalized routine. You'll have to learn assembly language to deal with this properly; use WSYNC, then rapidly store up to three colors after the WSYNC using STX, STY, and STA. You'll still be offscreen. For those of you I've lost, the timing of a DLI routine is a very touchy thing; if you don't know machine language and how the Atari relates to the TV, forget it.

This routine will also crash in graphics 8 as it will not complete between interrupts if you have interrupts on two consecutive scan lines. If you want that, learn assembly language, then write your own driver.

On Memory Boards

Q: My Atari dies after being on for a while. Or, my Atari freaks out unexpectedly.

Listing 1.

```
10 REM GRAPH PROGRAM
20 REM LAYOUT:
30 REM
40 REM 1 LINE GR.2       20 BYTES 16
50 REM 1 LINE GR.2       40       32
60 REM 1 LINE GR.0       80       40
70 REM 120 LINE GR.8 80+(4800)    160
80 REM 1 LINE GR.0       +40      168
90 REM 1 LINE GR.0       +40      176
100 REM 1 LINE GR.0      +40      184
110 REM 1 LINE GR.0      +40      192
200 REM SET MODE
210 GRAPHICS`8+16:REM FAKE LAST FOUR
220 REM DISPLAY LIST
230 ST=PEEK(560)+256*PEEK(561)
240 REM ST+0,ST+1,ST+2=112..LEAVE BE
242 REM ST+3=79. CHANGE TO 7+64.
243 POKE ST+3,7+64
245 REM ST+4,+5=DATA. LEAVE BE.
246 REM ST+6,ST+7=15. MOD TO 7,2.
247 REM (MODE 2, THEN MODE 0).
248 POKE ST+6,7
249 POKE ST+7,2
250 REM DM + 0 - DM + 29 = MODE 2 L1
255 GOSUB 1000
260 DIM A$(60)
261 SETCOLOR 4,8,2
270 A$=" MODE 2 BIG TITLE   "
280 REM 12345678901234567890
290 REM TRANSLATE A$ TO INTERNAL CSET
300 GOSUB 500
330 REM FIND DISPLAY MEMORY
340 DM=PEEK(ST+4)+256*PEEK(ST+5)
410 REM POKE INTO MEMORY
420 FOR T=1 TO 20
430 POKE DM+(T-1),ASC(A$(T,T))
440 NEXT T
450 A$=" MODE 2 SECOND LINE "
460 REM 12345678901234567890
470 REM TRANSLATE A$ TO INTERNAL
    CSET
480 GOSUB 500
485 REM POKE INTO MEMORY
490 FOR T=1 TO 20
493 POKE DM+20+(T-1),ASC(A$(T,T))
496 NEXT T
497 GOTO 600
500 REM SUBROUTINE TO XLATE ASC TO
510 REM INTERNAL CSET
520 FOR Z=1 TO LEN(A$)
530 IF A$(Z,Z)=" " THEN A$(Z,Z)=
    CHR$(0)
540 IF ASC(A$(Z,Z))<>0 THEN A$
    (Z,Z)=CHR$(ASC(A$(Z,Z))-32)
550 NEXT Z
560 RETURN
600 REM DO MODE 0 LINE NEXT.
     40 BYTES
610 A$="  A TEXT MODE 0 SUBTITLE"
620 REM XLATE
630 GOSUB 500
640 REM POKE INTO MEMORY
650 FOR T=1 TO LEN(A$)
660 POKE DM+40+(T-1),ASC(A$(T,T))
670 NEXT T
675 REM PLOT A SAMPLE GRAPH
676 SETCOLOR 2,8,0
680 XMIN=2
690 YMIN=5
700 XMAX=319
710 YMAX=159
720 COLOR 1
725 PLOT 1,70:DRAWTO 319,70:PLOT 1,70
726 XSAV=1:YSAV=70
730 FOR X=5 TO 315 STEP 5
740 Y=INT(RND(0)*70)+40
750 DRAWTO X,Y
752 PLOT XSAV+1,YSAV:DRAWTO X+1,Y
753 PLOT XSAV+2,YSAV:DRAWTO X+2,Y
755 XSAV=X:YSAV=Y
760 NEXT X
770 REM PUT IN 4 TEXT LINES AT BASE/
780 REM AFTER 160 (GR.8) INSTRUCTIONS
790 GOTO 790
1000 FOR Y=ST+150 TO ST+210
1010 IF PEEK(Y)=65 THEN 1100
1020 NEXT Y
1030 PRINT "PLATO OFF."
1040 STOP
1100 B1=PEEK(Y+1)
1110 B2=PEEK(Y+2)
1120 POKE ST+162,B2
1130 POKE ST+161,B1
1140 POKE ST+160,65
1150 RETURN
```

Or, my Basic programs scrozzle themselves. Or....

A: 1. If you squeeze the last few bytes of available memory, Basic seems to screw up. Something in the upper memory management routines fails during tight squeezes, and there isn't much you can do about it.

2. The Atari memory boards may be giving you trouble. Here's Small's Memory Board Fix (which works amazingly often on bizarre Atari problems):

The Atari memory boards get hot, really hot, in their enclosed metal cans in the enclosed metal cage. This heat can mess things up, particularly in the connectors. The metal is necessary to avoid spraying radio frequency interference all over, but it does cause problems. So every month or so we pull all the boards out of the Atari and re-seat them. This re-establishes the socket connection. Cleaning the ends of the connector (a pencil eraser works wonders) and coating them with Lubriplate, then re-seating them is also a good idea—helps prevent corrosion.

If this fixes it, fine. If not, go the drastic route (as we had to on one very touchy 800):

1. Remove the lid. Bypass the interlock with a taped in Q-Tip.

2. Remove the memory board lids (pull the two Phillips head screws). Re-install the boards.

This will really help to keep things cool. Of course, you may not be able to watch TV nearby (nor will your neighbors) but it *will* prevent overheating.

Now that you have the lid open, some of you are doubtless going to get the clever idea of copying ROM cartridges onto disk. After all, you can boot up, then plug them in with DOS running. Then, a simple binary save, right?

Wrong.

Atari has some nasty, nasty surprises awaiting you if you try this. First, plugging the cartridges in sends a nice hefty spike into the memory lines, straight into sensitive Antic, CTIA, and the 6502B. Do you really want your Atari in the repair shop? All it takes to destroy these chips is a little static electricity in the wrong place, and your body is probably full of it in the winter.

Second, the Atari people have some special checks to prevent this. For example, disk I/O doesn't work the way you might expect from cartridges. Ever had your directory mysteriously disappear? This should be food for thought.

On Piracy

Speaking of piracy in general. I have found copies of my software (what goes into these articles) floating around all over the place. This is really embarassing when the disk that was pirated is a development disk and you've saved all sorts of junk on it.

But second, when you think about it, the prices you pay for software nowadays in many cases are pretty low anyway (when was the last time you could go on a date for $20), so why not give the author his royalties, and get the documentation as well?

I wish that people didn't consider protection schemes a Scott Adams adventure #30 to be broken. If you think about it, the hours you spend breaking the scheme are equal in dollars to what you would pay for the software in many cases. (And if you're thinking about selling copies, don't; all the software companies I've talked to are *currently* prosecuting people caught doing this.)

Q: I have 32K. Should I get 48K?

A: Maybe. If you use no cartridges, the Atari can use up to 48K RAM. If you use one cartridge, you are reduced to 40K available; if you use two, 32K. Eventually, as more RAM-only programs become available, 48K will be more and more handy. For example, Microsoft Basic, which we are currently testing, requires 48K but has no cartridges (disk based). We're in a transition period, in other words, and it may be to your advantage to wait a bit; hardware prices are dropping quickly, as usual.

On Disks

Q: During a disk access, my disk stops for a while for no reason and then restarts. Why?

A: A bug in the O.S. program. No, the disk isn't stopping to cool off (like an 820 printer) or anything. This is fixed in the new revision cartridges, which are slowly becoming available.

Q: What are DOS 2.5, 2.7, 2.8, 2S, 2.0S, 2.0D?

A: DOS 2.0S is the final, "cast in concrete" version of DOS 2. The others are developmental versions. They are pre-release copies. There are lots of 2S disks lying around; these have a bug in the interrupt subsystem, so best get rid of them. Also, if you boot up under 2S, you can't "DOS" to a 2.0 version of DOS. They're incompatible. So your best bet is to change your disks over to 2.0S and use it.

DOS 2.0D is for the double density 815 drive, which has been cancelled, delayed, sent back, or whatever (depending on who you talk to).

Q: What is a "fast formatted disk?"

A: Inside the 810 disk drive there is a microprocessor. When the Atari wants a given disk sector (128 bytes), it asks that microprocessor for it. The micro then spins the disk and moves the head to get that sector. If you have a disk with a more efficient layout, you can go between sectors (without a complete spin between them, for instance). A "fast formatted disk" has this improved layout, and, thus, when you access it, disk I/O is around 20% faster.

Disks that you format with your 810 will not have this improved layout, because it lays them out the old, slow way. A new ROM, called the "C" ROM, can be installed into your disk drive to make it format disks the fast way.

Who knows when it will be available? The rumour mill says that 1) all disk drives going to Europe have it; 2) all disks to the East Coast have it; 3) all disks shipped after September 1981 will have it, etc. Probably by the time this is printed some policy will have been established.

For those of you who can't wait, the Chicago area user's group has constructed their own version of the format ROM, which requires a few wiring changes to the disk and programming a new EPROM (not your beginner-level stuff). The Chicago ROM is 10% faster than the Atari ROM, which is definitely interesting. The ROMs work quite well; I've seen them tested. However, since the Chicago folks developed them I'll let them document it and take the credit. Incidentally, modifying a drive this way (of course) violates the warranties.

On GTIA

Q: What's the GTIA chip and how do I get one?

A: The CTIA chip actually generates color for your TV. A new chip, GTIA, replaces CTIA and allows graphics modes 9, 10, and 11 out of Basic. (The operating system was written with GTIA in mind, and so was Basic, by the way.) It is an upgrade to the CTIA chip. The rumour mill again says it is available everywhere except where the rumour originates. We have one as the result of extreme kindness on the part of Atari, and are testing it. The added modes are:

Graphics 9: Allows 16 intensities (select by COLOR #) of pixels to be displayed in the background color. Great for grey-scale shading.

Graphics 10: Allows eight different kinds of pixels to be displayed in any of the standard colors. Uses the four P-M registers and four playfield registers to set colors.

Graphics 11: Allows 16 different colors for pixels, all in the same intensity.

The pixel size is four bits long, and one scan line high. This is 80 x 192 resolution, an interesting twist on the general rule that vertical resolution is less than horizontal.

There will be a more complete article on the GTIA chip when it is more widely available. (The problem is, most people at Atari don't have them either, and are trying just as hard to get one. Who do you think will get priority?)

On Languages

Q: Forth?

A: Forth is a dynamite programming language available for the Atari. Its speed is somewhere between Basic and assembly language, but much closer to assembly language. Best of all, it's a reasonably high level language (very stack oriented, as a matter of fact). I'm trying to learn it now. Versions are available from many sources. Atari lists Forth in their APEX exchange, but will not release it yet. Beware of other versions which may use undocumented entry points in the operating system, and which will quit working when the new cartridges are generally available.

There has been a lot of good software written in Forth. I have a synthesizer program, lent to me by Ed Rotberg of Atari, which plays the best music I ever heard from an Atari (and has different instrument sounds, too; drums, guitar, hand clapping, etc.). The Atari demo with the "Disco Dirge" is written in Forth to give you an idea of its execution speed and flexibility.

Q: Microsoft Basic.

A: You will be hearing a great deal about this from us. We are currently working with Microsoft Basic and it is a fantastic product, indeed. It is much faster than the Atari 8K cartridge Basic and has many, many more functions. It really turns the Atari into a serious business computer, for example. Look at the description of Microsoft Basic in any Apple, TRS-80 or PET book and you will get an idea what is available. Add to that many special Atari functions, and soon you will be writing only in Microsoft. □

Atari Languages

Opening the mailbox has become a bit like Christmas, with users sending in their latest code and accomplishments, plus new product announcements. Here's some of the best we have seen.

Drew Holcomb sends a very nice graphics demonstration (Listing 1) which is worth the five minutes it takes to type in.

Thomas Marshall (those of you on the CERL PLATO network system know him as marshall/phystemp) sends in the fine program in Listing 2 which uses the DLI routine from the December column. It puts a 128-color menu onscreen, then allows the user to move a cursor around the colors. When the user settles on a color and presses the button, the decimal value of the color (for use in SETCOLOR) appears in players on the top and bottom of the screen. There are some very nice techniques being used here; the program deserves a good look. Thanks Tom.

Dennis Baer (868 Main St., Farmingdale, NY 11735) has Algol for the Atari. According to his letter, it supports all I/O and graphics also. He also has a word processor for the Atari written in Algol. Since there are quite a few folks familiar with Algol, you might want to get in touch. He mentioned he is interested in beta-testing his product.

The Young People's LOGO Association wants to hear from people interested in Atari Pilot. Contact them at 1208 Hillsdale Dr., Richardson, TX 75081. They have a very good newsletter and a great deal of interest in the Logo language.

Atari in Europe

Finally, Nigel Haslock in Switzerland wrote to give me details of the European Ataris. Software houses may be quite interested in this information. He writes:

•European Ataris run 12% slower if tied to VBLANK.

•Atari has kept the one CPU clock/color clock; hence, the 6502 is a 3 MHz model (not 2 MHZ as in the United States), and is clocked at 2.217 MHz or about 25% faster.

•All European models have GTIA chips—hence the GTIA shortage here.

•The E000 and F000 ROMS are different (hence many software problems).

•Cassette handling is different and possibly incompatible.

I have also received a great deal of mail concerning piracy and disk copy protection, which Nigel mentions. He tells of not being able to fix US-version Atari programs to work on the European Ataris because of the copy-locks placed on them. He has a good point.

There is a European market looking for software. Besides the obvious language problems, software houses have another worry—will their software work as it is with a PAL TV?

Atari has provided a hardware location to determine if a given machine is PAL (European) or NTSC (North American); it looks as if it's time to start writing software to check it.

Atari Basic

For those of you with new Ataris, here is a short and highly opinionated discussion of the various languages available for your

Listing 1.

```
WHEELS
5 REM WHEELES WITHIN WHEELS BY D. HOLCOMB
10 GRAPHICS 23:Y=INT(RND(0)*16):FOR X=708 TO 711:POKE X,Y*16+12:NEXT X:COLOR 1:D
EG
20 FOR X=1 TO RND(0)*4+2:A=INT(RND(0)*5)*1000+2000:POKE 77,0
30 B=(INT(RND(0)*11)*INT(RND(0)*5)^2-79)*INT((RND(0)*2)-1)/(A/1500)+79
35 C=(INT(RND(0)*4)*INT(RND(0)*5)^2-47)*INT((RND(0)*2)-1)/(A/1500)+47
40 D=0:E=INT(RND(0)*3)-1:IF E=0 THEN 40
50 FOR Y=0 TO A STEP 15:F=F+E:IF F<1 THEN F=3
60 IF F>3 THEN F=1
70 COLOR F:D=D+0.08:DRAWTO B+D*COS(Y),C+D*SIN(Y)
80 NEXT Y:NEXT X
90 FOR Z=0 TO 9
100 A=RND(0)*16:B=RND(0)*9+4:C=RND(0)*30+10
110 FOR X=0 TO 1500/C
120 FOR Y=0 TO 2:SETCOLOR Y,A,B:FOR F=0 TO C:NEXT F
130 SETCOLOR Y,0,0:NEXT Y:NEXT X:NEXT Z:GOTO 10
```

use. It may serve to clear up some of the confusion you may have over which language is best for you to buy and use. Doubtless, there will be those who will disagree with me; feel free to write and let me know if you do.

Atari Basic, in the 8K cartridge is the original language for this system, developed in a great hurry for the unveiling of the new Atari machines back in 1978. Like most things done in haste, it lacks something. In this case, speed and the fixing of obvious bugs were neglected to the point where the whole product was compromised.

All arithmetic done in this Basic is done in 6-byte BCD. While this gives great accuracy, it also slows execution to a crawl. Atari 8K Basic is the slowest Basic I have ever used. To be sure, all computers have a tradeoff between memory use and speed, but this is a little ridiculous.

Add to this the many known bugs that will crash the machine, the slowness and occasional inaccuracy of the floating point operations, and numerous other flaws, and it just isn't much of a language. It could have been done better.

Unfortunately, so much of the available software uses the Basic, and even the bugs (remember the old saw about "documented bugs" becoming features in the next version), that Atari can't fix it. We're stuck with it. Too bad.

For speed reasons, it is just about impossible to write professional software in Atari Basic; any assembly program runs so much faster than the Basic that there is no comparison. Games written in Basic are easily identifiable by their slow speed.

In all fairness, the Atari Basic cartridge was meant for small Basic programs, not the huge amounts of code it is sometimes asked to execute, so it must be forgiven. The overall design structure is just wrong for fast program execution.

In conclusion, I wish it were better, but we are stuck with it. Great things have been done with the computer in spite of the Basic, and many Atari users have been forced to 6502 Assembler because of it.

Speaking of which...

Atari 8K Editor/Assembler

The 8K Editor/Assembler is a close relative of the Basic cartridge. It is absolutely unacceptable for major software development. I can remember delays of up to an hour assembling large programs, and I have heard many other horror stories. This cartridge is also 8K and has bugs.

For instance, any CPY instruction hangs the TRACE function. This cartridge has inspired many software houses to come

Listing 2.

```
COLOR128
19000 REM The following subroutine can
19001 REM be added to program 5 of
19002 REM ATARI OUTPOST in the December
19003 REM issue of Creative Computing.
19004 REM It addes utility to displaying
19005 REM the 128 colors available to
19006 REM the Atari.  By plugging a
19007 REM joystick in port 1, the sub-
19008 REM routine will give you the
19009 REM specific number one needs to
19010 REM poke in the color registers.
19011 REM
19020 REM POKE 708,XXX   COLOR 0
19021 REM POKE 709,XXX   COLOR 1
19022 REM POKE 710,XXX   COLOR 2
19023 REM POKE 711,XXX   COLOR 3
19024 REM POKE 712,XXX   COLOR 4
19025 REM
19030 REM In program 5, besure to add
19031 REM 410 GOTO 20000
19032 REM
20000 X=105:Y=8
20010 A=PEEK(106)-24:POKE 54279,A:PMBASE=256*A
20020 POKE 559,46:REM DOUBLE LINE RES.
20021 POKE 623,4:REM PLAYFIELD OVER PLAYER PRIORITY
20022 POKE 53277,3:REM TURN ON PM GR.
20030 FOR III=PMBASE+384 TO PMBASE+1024:POKE III,0:NEXT III:REM CLEAR P-M GR....
20040 POKE 53248,X:REM PL 0 POSITION
20041 POKE 53249,X+16:REM PL 1 POSITION
20042 POKE 53250,X+32:REM PL 2 POSITION
20043 POKE 53251,X-32:REM PL 3 POSITION
20044 POKE 53252,X+48:REM MI 0 POSITION
20045 POKE 53253,X+56:REM MI 1 POSITION
20046 POKE 53254,X+64:REM MI 2 POSITION
20047 POKE 53255,X+72:REM MI 3 POSITION
20050 POKE 704,53:REM PL 0 COLOR ORANGE
20051 POKE 705,65:REM PL 1 COLOR RED
20052 POKE 706,145:REM PL 2 COLOR BLUE
20053 POKE 707,27:REM PL 3 COLOR YELLOW
20069 REM PLAYER SIZE:  0=NORMAL,1=DOUBLE,3,DQUADRUPLE
20070 POKE 53256,1:POKE 53257,1:POKE 53258,1:POKE 53259,3
20071 POKE 53260,255:REM SAME CONVENTION BUT 2 BITS FOR EACH MISSILE SIZE
20080 REM *** PRINTS MISSILE NOS. *****
20090 CHR=17:II=0:FOR III=PMBASE+512+Y TO PMBASE+519+Y:POKE III,PEEK(57344+CHR*8
+II):II=II+1:NEXT III
20091 CHR=18:II=0:FOR III=PMBASE+640+Y TO PMBASE+647+Y:POKE III,PEEK(57344+CHR*8
+II):II=II+1:NEXT III
20092 CHR=24:II=0:FOR III=PMBASE+768+Y TO PMBASE+775+Y:POKE III,PEEK(57344+CHR*8
+II):II=II+1:NEXT III
20093 FOR N=PMBASE+512 TO PMBASE+1024 STEP 128
20094 FOR M=16 TO 0 STEP -1
20095 POKE N-M,255:NEXT M:NEXT N
20096 XO=125:XN=125:YO=93:YN=93:COLR=255:Y=8:GOTO 20170
20099 REM ***** JOYSTICK ROUTINE ******
20100 MOVE=STICK(0):IF STRIG(0)=0 THEN 20230
20105 IF MOVE<>7 AND MOVE<>11 AND MOVE<>13 AND MOVE<>14 THEN 20100
20110 IF MOVE=14 THEN YN=YO-3:COLR=COLR-2:IF YN<0 THEN YN=93:COLR=COLR+64
20120 IF MOVE=13 THEN YN=YO+3:COLR=COLR+2:IF YN>94 THEN YN=0:COLR=COLR-64
20140 IF MOVE=11 THEN XN=XO-40:COLR=COLR-64:IF XN<0 THEN XN=125:COLR=COLR+256
20150 IF MOVE=7 THEN XN=XO+40:COLR=COLR+64:IF XN>141 THEN XN=5:COLR=COLR-256
20170 LOCATE XO+10,YO,CO:CN=CO-2:IF CN<0 THEN CN=CO+2
20180 COLOR CO:PLOT XO,YO:DRAWTO XO+5,YO:PLOT XO,YO+1:DRAWTO XO+5,YO+1
20190 COLOR CN:PLOT XN,YN:DRAWTO XN+5,YN:PLOT XN,YN+1:DRAWTO XN+5,YN+1
20200 XO=XN:YO=YN
20220 GOTO 20100
20230 FOR N=0 TO 3:POKE 704+N,COLR:NEXT N
20235 REM DISASEMBLE COLR FOR PM PRINT
20240 IF COLR>99 THEN CHR0=16+INT(COLR/100):CHR1=16+INT((COLR-(CHR0-16)*100)/10)
20241 IF COLR>99 THEN CHR2=16+COLR-(CHR0-16)*100-(CHR1-16)*10:GOTO 20300
20250 IF COLR>9 THEN CHR0=16:CHR1=16+INT(COLR/10):CHR2=16+COLR-(CHR1-16)*10:GOTO
 20300
20260 CHR0=16:CHR1=16:CHR2=16+COLR
20300 II=0:FOR III=PMBASE+512+Y TO PMBASE+519+Y:POKE III,PEEK(57344+CHR0*8+II):I
I=II+1:NEXT III
20310 II=0:FOR III=PMBASE+640+Y TO PMBASE+647+Y:POKE III,PEEK(57344+CHR1*8+II):I
I=II+1:NEXT III
20320 II=0:FOR III=PMBASE+768+Y TO PMBASE+775+Y:POKE III,PEEK(57344+CHR2*8+II):I
I=II+1:NEXT III
20500 GOTO 20100
20890 REM
20900 REM If you change graphics modes
20905 REM from here, suggest you
20910 REM FOR N=0 TO 7
20920 REM POKE 53248+N,0
20930 REM NEXT N
20940 REM to remove the player-missile
20950 REM graphics from the screen.
```

out with their own assemblers, some of which are very good. The Atari cartridge has a handy debugger, but as an assembler, (how do I put this tactfully?) it is useful only for assembling small subroutines for Basic.

One good thing about the Basic and Assembler/Editor cartridges is that there are now books designed to help the beginner get going with both of them. This is a good way to learn about the Atari and to get started but don't limit yourself to these products once you are past the beginning stage. Another advantage is that they are cartridges, so you don't have to buy a disk drive or more memory in order to run them, as is the case with most other languages.

The above products are supported by a company called Optimized Systems Software, located in Cupertino, CA. OSS also markets a 16K disk-based Basic known as Basic A+ and an assembler in their operating system called EASMD. Lo and behold, Basic A+ and EASMD are so close to the original cartridge code that they even have the same bugs.

The Basic is just as slow, but it does have new commands for handling players and missiles and disk I/O. It is a big step up from the original Basic, but still needs work. I would like to see integer variables and something to speed up the execution?

Microsoft Basic

Atari Microsoft Basic is the Basic that Atari should have released initially. It is a 19K disk-based Basic. Add to that about 8K of DOS which must also be booted with it, and the result is 21K of user memory available for the programmer on a 48K system or only 13K on a 40K system.

That could be hard to live with. For instance, if you go into graphics 8, you have only 13K left on the 48K system, and only 5K on the 40K system. Still, Microsoft Basic is a very powerful and convenient language to use, and I have found few bugs in it. But don't get it unless you have 40 or 48K.

Microsoft Basic has integer variables which are very fast, PRINT USING for business applications, and 4- or 8-byte accuracy (whichever you select), which speeds everything up. It is very much like TRS-80 Basic or Applesoft. Best of all, it has several nice features for player missile graphics, character set redefinition, and other Atari-specific capabilities.

I like it, and try to use it whenever I have to do anything serious in Basic. While the bootup process takes a while, the time saved in program development is worth it. Any professional developer should seriously consider Microsoft Basic.

A 16K single cartridge (yes, you can put 16K on a single cartridge, check the hardware manual for details) version of Microsoft Basic is planned, but some features, such as renumber or PRINT USING, may still have to boot in from disk. More on this later; things haven't settled down yet.

Microsoft has extremely good documentation which looks even better when compared to the original Basic document. It was this documentation that was responsible for the delay in delivering the product; the disks have been ready to ship for some time but the manuals weren't. Considerable time and effort have gone into them, and it shows. Good job.

Atari Macro Assembler/Editor

The Atari Macro Assembler/Editor is a very, very powerful disk-based assembler, which is a joy to work with. Light years beyond the original cartridge, it is extremely fast; it will completely assemble 100 pages of code in six minutes. It features support for independent files with Include, macros, systext files, and raw speed. I have discovered a few trivial bugs in it, but this is one product I can rave about without reservation. I have worked with it for more than four months and like it better each time I use it.

If you have any serious assembly language programming to do, get the Macro Assembler/Editor.

The Editor is also quite nice, and is being sold through APEX. It is a powerful and reasonably fast editor for developing text with no line numbers. This allows easy input of data, since you need not bother to strip off extraneous line numbers. Pascal, the assembler, and future goodies rely on the editor to generate source text.

Atari Pascal

Atari Pascal is brand new, and not reviewed yet. It is not a UCSD Pascal, but those of you who like Pascal might want to look it over ˙and send me comments.

Atari Pilot

The Atari version of Pilot is a pretty clean implementation of the famous educational language. I have not done much with it but the feedback I have gotten is all good. The documentation in particular is extremely well done.

Forth

To understand "Why Forth?" you must look at some basic programming philosophy. Many languages are unsuitable for serious software development work. For most high speed games, for example, even Microsoft Basic isn't fast enough. For business applications Pilot is out, and so on.

Well, Forth is difficult to describe, but let me try: it is a stack-oriented language which you define yourself. You start with a basic set of commands (input, output, arithmetic), and define your own commands (called words) from there. The language executes extremely quickly, compared to everything except assembler, and once you get into it, is much easier to write and debug, which drastically cuts development time.

Assembly language provides the ultimate in speed and machine control, but is not much fun to work with. Even with the very good macro assembler, debugging assembly code (especially without very good debug tools) is a frustrating, time consuming process. Forth helps the user get away from that.

The Atari is a very good machine for Forth. There are so many unique hardware features that a generic language such as Basic isn't good at handling them all, but Forth is.

You can define language commands to deal with players and missiles, character sets, vertical blank interrupts, and whatnot. Each user's Forth thus ends up growing along with him.

Sandy and I had been thinking of going to Forth, but it seemed like too much effort to get started. There were even two Atari Forths on the market: QS Forth and Pink Noise Forth. Yet I had a difficult time following QS Forth, even with its reasonably good instructions, and it seemed more a generic Forth implementation than an Atari-specific Forth. So we waited.

Two things changed our minds: 1) The book *Starting Forth* by Leo Brodie, from Prentice-Hall, which is simply superb and easy to read (complete with really, really good illustrations), and 2) VALForth, a new Forth based on figForth, which is currently being sold by APEX.

VALForth has commands designed especially to take advantage of the features of the Atari. It also includes a character set editor, easy player graphics, a very nice screen editor, and several other useful features. We were shown a preliminary version, and after reading the *Starting Forth* book, could dive in immediately and do things.

The transition is reasonably painless, and the power of the language unfolds around the user; I'm very happy to be working in it and we plan to write our next game in VALForth. The stack orientation is easy to get used to, especially if you just consider the stack data as part of the instruction set format.

The development speed of Forth has not been overlooked by Atari. Atari's Coin-Op group has a semi-legendary "Coin-Op Forth" which is supposed to be quite something to use. The Atari demo disk with the "Disco Dirge" background music is all "coin-op Forth." Rumor also has it that many of the new Coin-Op games are written in Forth (Battlezone, for one). This wouldn't surprise me; it's a powerful language.

As a point of philosophy, we feel that as Atari programmers we began in Basic, moved to 6502 Assembly for speed, and now, after experiencing assembly debugging, are moving to Forth to reduce the amount of time we must spend on programming. We have great hopes for VALForth and what we have seen already is very worthwhile.

Getting Along Without TAB — An Atari Translation

Fred Pinho

The lack of a TAB command in Atari Basic is a source of irritation to many Atari users. The most common problem occurs when outputting formatted text; cumbersome programming is necessary to accomplish what is relatively simple in other Basic dialects.

Listing 1.

```
40 B1=20
50 B2=B1+10
60 T=24
80 REM
90 GET C$
100 IF C$=" " THEN S=S+1:IF S<2
    THEN 90
101 S=0
120 IF C$=<>"1" AND C$<>"2" THEN 140
130 ON VAL(C$)GOSUB 220,230
140 IF (B1=T) OR (B2=T) THEN 400
150 PRINT TAB(B1);D$;TAB(T);"+";TAB
    (B2);D$
160 A=INT(RND(1)*5)+1
170 ON A GOSUB 200,210,210,210,200
180 ON B GOSUB 250,240
190 M=M+1:GOTO 80
200 B=1:RETURN
210 B=2:RETURN
220 T=T-1:RETURN
230 T=T+1:RETURN
240 GOSUB 350:RETURN
250 Y=INT(RND(1)*3)+1
260 IF X=Y THEN 250
270 X=Y
280 IF X=1 THEN D$="/"
290 IF X=2 THEN D$="I"
300 IF X=3 THEN D$="\"
310 GOSUB 350
320 RETURN
350 B1=B1+X-2
360 IF B1<1 THEN B1=1
370 B2=B1+10
380 IF B2>39 THEN B1=29:GOTO 370
390 RETURN
400 PRINT TAB(T);"*    CRASH!!!"
410 PRINT "YOU SCORED ";M; " POINTS."
420 M=0
430 FOR I=1 TO 500:NEXT I
460 GOTO 40
```

Fred Pinho, 676 Rollingwood Way, Valley Cottage, NY 10989.

It can also be a problem in other areas such as games where a given character must be printed at varying locations on a line. To illustrate this, the car race program shown in Listing 1 was translated into Atari Basic. This program originally appeared in the November 1980 *Creative Computing* as a translation from DEC PDP/11 to PET Basic. The game depends on the printing of the walls of the road and of the car under control of the TAB command.

The Atari does have a keyboard-controlled tab function which can be used in the programming mode by printing it in properly configured strings. However, running the game in that manner would be difficult. Fortunately there is another way.

The Atari does not print at the keyboard-set tabs unless specifically requested to do so by an imbedded tab request within the string to be printed. Rather it prints at standard "print positions" positions 0, 11, 21, 31 on a 38-character line).

Separating the desired strings by a comma causes each string to be printed starting at a standard position. The width between each print position is controlled by memory location 201. Don't be fooled by its name in the Atari reference manual. Although it is called the "Print Tab Width," it really controls the width of the print positions (sneaky).

The Atari translation is shown in Listing 2. The parameters to be used in controlling the width of the print positions are in line 40: B1 (left side of the road), B2 (right side of the road) and T (the car). The actual printing is controlled by lines 150-153. Here location 201 is POKEd with the width for the left side of the road. Then printing a blank followed by a comma spaces the invisible cursor to the second print position (controlled by B1). D$, which forms the sides of the road, is then printed.

Since the cursor has now moved down to the start of the next line, location 84 (current cursor row) is decremented by 1

Listing 2.

```
1 REM ATARI TRANSLATION BY FRED PINHO
2 REM FROM PET TRANSLATION BY D. LUBAR  AND R. FORSEN
3 ? ">"
5 DIM D$(1),A$(3)
10 POKE 752,1
20 S=0:M=0
40 B1=20:B2=B1+7:T=24
45 ? "        INDIANAPOLIS SPEED TRIALS"
50 ? "          SPEED DEMONS WANTED"
60 ? "    ARE YOU WILLING TO GIVE IT A TRY";:INPUT A$
70 IF A$<>"Y" AND A$<>"YES" THEN END
80 ? :? :? "PRESS N TO GO LEFT,M TO GO RIGHT,"
81 ? "AND SPACE TO GO STRAIGHT":FOR P=1 TO 2000:NEXT P
85 ? ">"
90 IF PEEK(764)=255 THEN S=S+1:IF S<2 THEN 90
100 S=0
120 IF PEEK(764)<>33 AND PEEK(764)<>35 AND PEEK(764)<>37 THEN 140
129 I=PEEK(764)-32
130 ON I GOSUB 235,235,220,220,230
140 IF B1>=T OR B2<=T THEN 400
150 POKE 201,B2:? " ",D$
```

to cause a return to the original line. The procedure is now repeated with the width set for the car (line 152). Here a graphics heart is used for the car. It doesn't show on the listing so type control-comma between the second pair of quotes in line 152. The procedure is repeated once more for the right side of the road.

The keys N, M and space were used to move the car. Rather than "opening" the keyboard and using a GET command, memory location 764 (last keyboard key pressed) was used. If you PEEK this location, you'll find an entirely different character set code is used instead of the one detailed in the Atari manual. This code is read and converted for use in lines 120-129. Finally, line 140 checks whether the car has collided with the side of the road. If so, it branches to the end-of-race routine.

Once I had the program working properly, like most programmers, I could not resist the urge to improve and upgrade it. What better way than to make use of the built-in sound and color capabilities of the Atari. The sound of a race car was easy (line 154) since a distortion level of 2 in the SOUND statement gives a very realistic sound.

For the inevitable crash (Mario Andretti I'm not), I turned to the January 1981 issue of *Creative Computing* for a "percussive sound generator." Modifying the explosion routine slightly worked well (lines 403-408).

```
152 J=PEEK(84):POKE 84,J-1:POKE 201,T:? " ",""
153 J=PEEK(84):POKE 84,J-1:POKE 201,B1:? " ",D$
154 SOUND 0,70,2,7
160 A=INT(5*RND(1))+1
170 GOSUB 250
190 M=M+1:GOTO 90
220 T=T-1:RETURN
230 T=T+1:RETURN
235 T=T:RETURN
250 X=INT(3*RND(1))+1
280 IF X=1 THEN D$="/"
290 IF X=2 THEN D$="|"
300 IF X=3 THEN D$="\"
350 B1=B1+X-2
360 IF B1<1 THEN B1=1
361 IF B1>26 THEN B1=26
370 B2=B1+7
390 RETURN
400 POKE 201,T:? "","*":? "CRASH!!!":SOUND 0,0,0,0
401 FOR K=1 TO 10:FOR I=1 TO 10:SETCOLOR 2,I,14:NEXT I:NEXT K
402 SETCOLOR 2,9,4
403 NTE=200:GOSUB 405:SOUND 1,0,0,0:SOUND 2,0,0,0
404 GOTO 410
405 SOUND 2,75,8,15:ICR=0.79+7/100:V1=15:V2=15:V3=15
406 SOUND 0,NTE,8,V1:SOUND 1,NTE+20,8,V2:SOUND 2,NTE+50,4,V3
407 V1=V1*ICR:V2=V2*(ICR+0.05):V3=V3*(ICR+0.08):IF V3>1 THEN 406
408 SOUND 0,0,0,0:RETURN
410 ? "YOU SCORED ";M;" POINTS."
411 IF M<=20 THEN ? "TRY AGAIN WITH A SLOWER CAR"
412 IF M>20 AND M<50 THEN ? "YOU'RE GETTING BETTER.KEEP PRACTICING!"
413 IF M>=50 AND M<80 THEN ? "YOU'RE A HOT ROD!"
414 IF M>=80 THEN ? "WOW!!! LET'S GO TO THE DRAG STRIP!!"
420 FOR I=1 TO 700:NEXT I:POKE 764,255
430 ? "}":GOTO 20
```

For the visual portion of the explosion, a simple rapid rotation of the screen colors was effective (lines 401-402). Note that the SETCOLOR had to be reset in line 402 to return the screen to the original color.

Finally I added a crude scoring system (lines 410-414) and a method of playing repetitively under player control. In line 420, location 764 had to be POKE with 255 otherwise the last direction key pressed, prior to a crash, would be printed after line 60 was executed.

If you get too good for the program, reduce B2 in line 40. Happy racing! □

Telecommunications and Memory Locations

John Anderson

It has been nearly three years now since I first unboxed my Atari 800, and I can report that it has performed unfailingly over all that time and heavy use. I'm not saying I have never had a system crash or lock-up, but the computer has never required any service beyond cleaning the board contacts once in a while (I can handle that).

The machine was and remains an engineering triumph, which is still ahead of its time in terms of capability and cost, as well as reliability.

Atari owners have had to be a tough-skinned breed now and then over the past three years, but that has changed. The "big three" are under the gun. Atari owners, now numbering over 300,000, know they made the right decision.

I am a member of this group, and a satisfying facet the hobby offers to me is the use of my machine to communicate with others who feel the same way I do about it: that the Atari is the best machine of its class, and that learning its secrets is an extremely pleasurable pastime.

Last May we instituted a call for reviewers, and many Atari owners responded. One thing that impressed all of us here at the magazine was our query concerning modems. Of the respondents who did not already use their microcomputers for telecommunication, nearly *everyone* responded that he had a modem on his "wish list," and that it wouldn't be long before he was hooked up. We have also had a very favorable response to the possibility of making *Creative Computing* downloads available over networks such as Compuserve and The Source. We are looking into this possibility.

The Modem Mystique

A great deal of potential presents itself. The possibilities of travel reservations,

John Anderson is an associate editor for *Creative Computing* magazine.

ticket purchases, shop at home services, a broad range of databases at your finger tips, are very exciting. Telecommunications herald a truly practical role for the microcomputer in the home.

I do not believe, however, that any of these practical notions constitutes the real basis for the "modem mystique." The thing that excites most people about microcomputer telecommunication is the opportunity to express themselves in a new medium, to tell others how they feel. They are less interested in using a modem to pay their bills than to state their opinions, to have their voices heard, and to respond to the voices of others.

The bulletin board service is growing in popularity. This is a phone line tied to a computer, running a program that accepts and displays information sent from other computers. The concept of the bulletin board is powerful and extensible. It creates a new kind of forum—a medium of communication—through which ideas can be expressed, shot down, modified, and spread. The importance of this kind of interaction, and its potential, is now being discovered. I think it may be a while before it emerges as a medium of major influence, but it *is* going to happen; it's happening now.

I maintain contact with about five Atari bulletin boards regularly. I enjoy leaving messages as well as reading what others have to say. I check the download files to see if there is any software worth trying out. I find out what other Atari owners are thinking about, as well as expressing my own thoughts. I may even start a "real-time" conversation with someone at the other end.

You may be a new user with questions concerning hardware. You may be an assembly language programmer wishing to share the results of a routine you have developed. Or, you may simply wish to voice your opinions concerning *Tron* or *E.T.* or to boast of your latest score at *Zaxxon*. The bulletin board is a worthwhile place to do it. Your thoughts join other thoughts, in what amounts to a marketplace of ideas—ideas that are shared.

Communication via computer may seem at first to be rather impersonal, but this is not the case. Through what other medium might you become involved in a lengthy philosophical chat with a sysop (system operator) hundreds of miles distant and at three in the morning? It is almost like being able to call your own user's group meeting whenever the mood strikes you—and then adjourn it without muss or fuss. It is at once personal and yet distant: and therein lies its unique value.

So get that Atari of yours talking to other Ataris, the way you've planned.

The Forth Wave

A very hot topic on nearly every board I have logged onto lately is the Forth language. This language offers hope to folks frustrated by the slowness of Basic, limitations of Pilot, bugs in APX Pascal, and obscurity of assembler. Although it has its own unique little quirks, Forth seems to be a natural for the Atari machine.

There are many implementations of the language available for the Atari, but the definitive version now seems to be Valforth, from Valpar International, 3801 E. 34th Street, Tucson, AZ 85713. We have received four packages from them, each of which shows a high level of professionalism and promise.

Valforth is a debugged and improved version of APX Forth, and is available with a powerful screen editor and utility package; a player missile graphics package, character and sound editor; and a display list formatter. We were able to create very smooth multicolored player/missile animation as well as modified display lists with very little fuss. The speed of movement is not as fast or as smooth as machine code, but is many times faster than Basic, and quite acceptable—so are the ease with which these packages can be used, and their reasonable cost. I hope to present a thorough evaluation of all the Valpar packages in the near future.

Despite whatever you may hear to the contrary, you need not renounce your

Other Atari Bulletin Boards

R=Ringback L=Limited service

Type	Name	Location	Phone
AMIS	—	Atlanta, GA	404-252-9438
AMIS	APOGEE	Miami, FL	305-238-1231-RL
AMIS	—	Baton Rouge, LA	504-273-3116
AMIS	ARCADE	Detroit, MI	313-978-8087-R
AMIS	—	Chicago, IL	312-789-3610
AMIS	GRASS	Grand Rapids, MI	616-241-1971
AMIS	MACE	Detroit, MI	313-868-2064
AMIS	MLBBS	Madison, WI	608-251-8538
AMIS	SB-12	Boston, MA	617-876-4885-L
AMIS	SPACE	Seattle, WA	206-226-1117
AMIS	TEAM	San Jose, CA	408-942-6975-L
ARMU	ARMUDIC	Washington, DC	202-276-8342
ARMU	FLEGLG	New York, NY	212-598-0719-L
ARMU	GREKLCOM	Oklahoma City, OK	405-722-5056
ARMU	PACE	Pittsburgh, PA	412-655-3046
ATBBS	—	Honolulu, HI	808-833-2616
TARI-BOARD	—	Denver, CO	303-221-1779
TARI-BOARD	—	Atlanta, GA	404-252-9438
CBBS	CP/M	Detroit, MI	313-759-6569-R
RBBS	CP/M	Allentown, PA	215-398-3937
RBBS	CP/M	Chicago, IL	312-789-0499

The "type" of bulletin board indicates what program is run on the host computer. Each program has its own strong and weak points; ergo each has its own adherents and detractors. It's all part of the fun.

AMIS stands for "Atari Message and Information Service." ARMUDIC began as a mnemonic for the phone number of the original service.

"Ringback" means to let the phone ring once, hang up, count to five, and redial. This allows a single line to serve as a voice and modem connection.

"Limited service" means the board is up only part-time, as opposed to 24 hours a day. If you can connect once, the hours will be listed for you.

These numbers were compiled by the MACE BBS, which is one of the most popular Atari boards in the country. Our thanks to the Michigan Atari Computer Enthusiasts for this list. Give them a call!

worldly ways to grasp Forth. Nor does mastery of Reverse Polish Notation cause hair loss or halting speech. Sure, the language has its peculiarities, but that's the challenge, right? Anyway, whenever you hit a real snag, you can ask for help on a bulletin board service! There's nothing like access to someone who knows the answers when you're trying to learn something.

Another byproduct of accessing user's group bulletin boards is the spreading of rumors. One such rumor I discovered on the MACE BBS has lamentably been confirmed: John Harris, brilliant young author of *Jawbreaker* and *Mouskattack*, had the only extant source code for his latest work, *Frogger*, stolen during a charity benefit. It is hard to understand what the thief had in mind—if we assume the thief had anything resembling a reasoning mind. What could he have hoped to gain by stealing the source code of an unfinished program? This will certainly forestall the release of *Frogger* for some months, and is sure to have put a real crimp into John's summer, if not his year. Upon capture, the thief should be forced to play Crystalware adventures to their solution or the thief's collapse, whichever comes first. (Are you taking any bets?)

Poking Around

I have yet to see a *definitive* list of memory locations for the Atari in any manual, periodical or book. We are compiling a list currently, and it will appear soon in the pages of *Creative Computing*. In the meantime, here is a very brief collection of some of the most interesting locations, and what values to POKE them with (all values in decimal):

65 - if = 0, I/O data transfer tones from TV or monitor will be disabled. Load will take place in silence. Nice with titles or especially music, to suppress "noise." If location 65 < >0, I/O will be audible.

77 - if = 0, attract mode will be suppressed. It is surprising to me how many programs are missing this simple POKE in any loop of less than nine minutes duration. Although designed to prevent "burn-in" on an unattended machine, this mode drives me nuts. If location 77 = 128, attract mode is enabled without nine-minute clock countdown.

752 - if = 0, makes the cursor "invisible." I say invisible rather than disabled because the cursor still functions as if it were visible. Nice in title cards and text programs to clean up screen "look." If location 752 < > 0, cursor will be visible.

82 - if = 0, enables 40-column screen width. The Atari defaults to a 38-character screen width, which was a good thing for me when I used a regular color television with the computer. "Overscan," as it is called, cut off the left-hand side of the screen. When I upgraded to a color monitor (much to my wife's relief), I noticed two unused columns on the left side of the screen. A simple POKE brings them into play. If location 82 = any number from 0 to 39, that number becomes the left-hand column. The default value is two.

83 - Same as above but for right-hand margin. Default is 39. Less call for this one, but nice to know, anyhow. Right?

Third party game software for the Atari 400/800 computers continues to pour in to the magazine. Let's take a look at just a part of the cream of the latest crop:

The Upstart Atari

John Anderson

June 4, 1981, *The New York Times* ran a relatively enlightened feature on the microcomputer and its future in the home. One of the "experts" cited in the feature stated the following: "there is almost no sense at all in buying a computer other than a PET, Radio Shack, or Apple." A bit further down the page, in a separate but allied article, the quote appeared *again*, this second time without the word "almost." The article referred to these companies as "the big three."

At the time, I was glad to see that the *Times* had discovered microcomputers, but was chagrined by what I saw as expert narrowmindedness. Still, it came as no surprise to me. I had acquired quite a stiff upper lip by that time. You see, I am an Atari owner.

I remember when I first began shopping seriously for a micro, right about the time the first Ataris were shipped. I had a great deal of trouble getting anyone to talk about the machine. Sales staff seemed so resentful in one computer store, I wondered aloud why they even carried the thing. A salesperson exclaimed to me, through a sneer, that he did not expect it would be carried for long.

Even as recently as a year and a half ago, finding an article concerning the Atari in a computer magazine was a triumph. The machines remained a mystery, even to those who owned them. Documentation and software were scant. I was told by more than one learned microguru that I had made an expensive error. They predicted nothing but early death.

John Anderson is an associate editor for *Creative Computing* magazine.

This was not to be. Despite the bad press and initial lack of documentation and software, the Atari was gradually discovered to be a superior machine: a "next-generation" micro, with ROM cartridge capability, a replaceable operating system, sophisticated color graphics capability, and four-channel sound.

Despite initial snobbery and snubbery, buyers began to opt for a good machine at a good price. By Christmas 1981, the Atari was being sold faster than it could be manufactured.

How did the competition respond to the introduction of the Atari? With the introduction of Atari lookalikes. Studying these, I realized Atari must have done something right, to have nearly everybody else shouting "me too!" within a year or so.

The Atari has been called a game machine, and games have certainly sold their share of units. Ted Nelson took a look at *Star Raiders* on a video projection system and proclaimed that the Atari Personal Computer was the "most extraordinary (microcomputer) graphics box ever made." Yet in addition, the Atari could do anything the "big three" could do, and then some. Many prospective purchasers found in the Atari a double bonus: a chance to have a "serious" microcomputer, while owning the greatest game machine around.

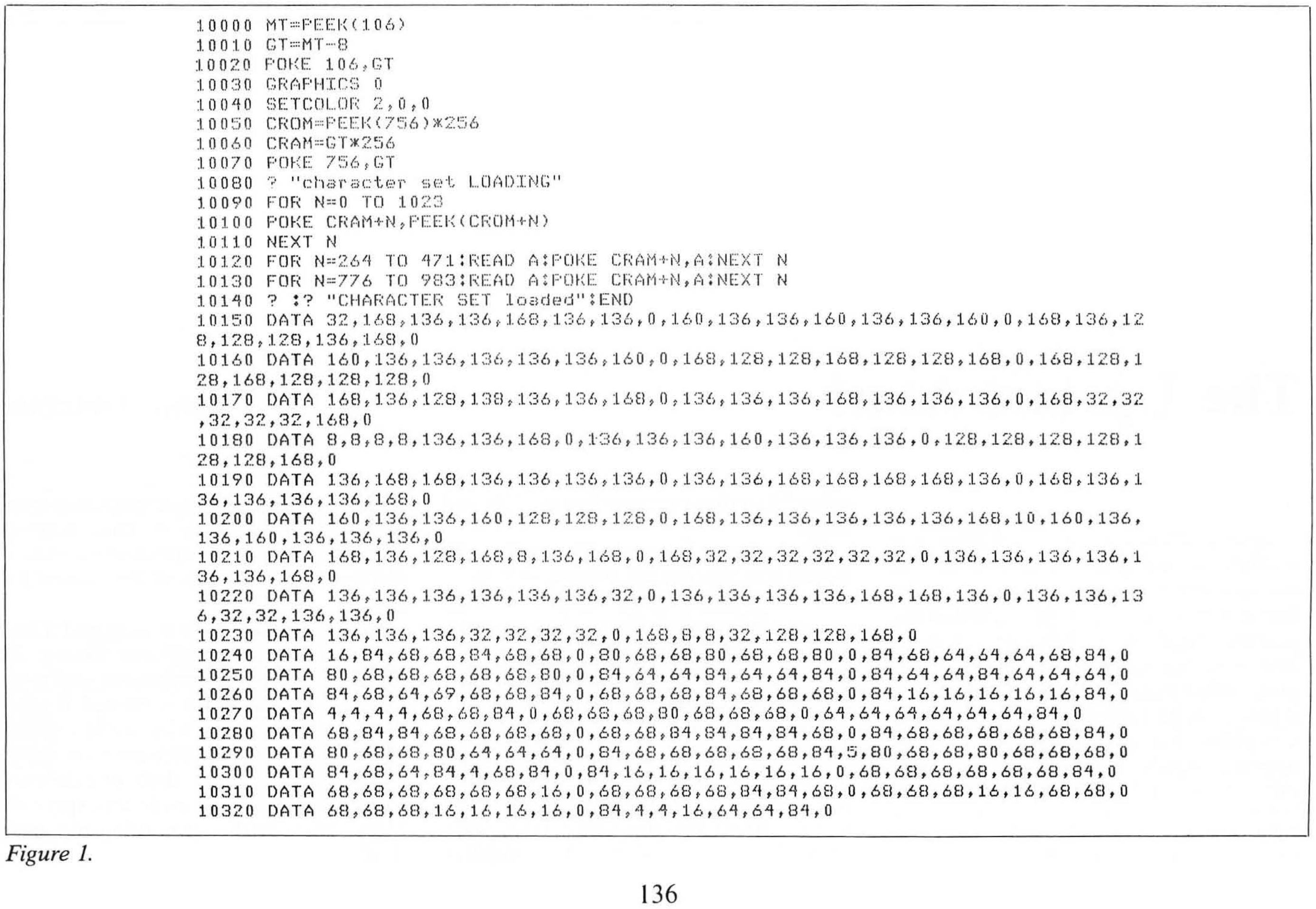
And, it was friendly. It is easier to do things right on the Atari, and more importantly, harder to do things wrong. The jargon terms machines like this "user-friendly." Never before had a computer been introduced that was so easy to use. Until the Atari came along, you couldn't expect to take a micro out of its box, plug it in, and have it work.

In the operating system of the machine is a powerful, built-in screen editor, which makes the mechanics of programming much less formidable on the Atari than on other machines. I know for a fact that this, combined with the syntax-checking function of Atari Basic, allowed me to learn Basic programming at a much faster pace than would have been possible with any other microcomputer. These features simply allow the user to recover more gracefully from his own errors, thus vastly increasing the utility of the machine as a learning tool.

Then there is the cost. I literally "paid the price" to be the first on the block with an Atari 800. Now, because of the popularity of the machine, prices have dropped dramatically. A bit of careful shopping can result in a basic unit for under $700. For this price, you receive an 800 with 16K of RAM and Atari Basic. The model 400 is down to about $250.

The computers have a built-in RF modulator, and so can be hooked directly to a home TV. A basic unit isn't worth much without cassette or disk storage devices, which constitute an additional expense, however the Atari disk drive has also been heavily discounted, and can be found for under $450. A 48K disk-based system can be put together for under $1400, and that is a good bargain at today's (and tomorrow's) prices.

As for the capabilities of such a system, let me first insert here a warning to those who may be unfamiliar with the moiling and sweaty world of microchauvinism. I feel strongly, as do other Atari owners, that a major part of what a microcomputer must handle superlatively is color graphics and sound. I take this to be a self-evident, foregone and unimpeachable tenet, and will make no effort to argue for or defend myself upon that point. If you do not concur, read on



```
10000 MT=PEEK(106)
10010 GT=MT-8
10020 POKE 106,GT
10030 GRAPHICS 0
10040 SETCOLOR 2,0,0
10050 CROM=PEEK(756)*256
10060 CRAM=GT*256
10070 POKE 756,GT
10080 ? "character set LOADING"
10090 FOR N=0 TO 1023
10100 POKE CRAM+N,PEEK(CROM+N)
10110 NEXT N
10120 FOR N=264 TO 471:READ A:POKE CRAM+N,A:NEXT N
10130 FOR N=776 TO 983:READ A:POKE CRAM+N,A:NEXT N
10140 ? :? "CHARACTER SET loaded":END
10150 DATA 32,168,136,136,168,136,136,0,160,136,136,160,136,136,160,0,168,136,12
8,128,128,136,168,0
10160 DATA 160,136,136,136,136,136,160,0,168,128,128,168,128,128,168,0,168,128,1
28,168,128,128,128,0
10170 DATA 168,136,128,138,136,136,168,0,136,136,136,168,136,136,136,0,168,32,32
,32,32,32,168,0
10180 DATA 8,8,8,8,136,136,168,0,136,136,136,160,136,136,136,0,128,128,128,128,1
28,128,168,0
10190 DATA 136,168,168,136,136,136,136,0,136,136,168,168,168,168,136,0,168,136,1
36,136,136,136,168,0
10200 DATA 160,136,136,160,128,128,128,0,168,136,136,136,136,136,168,10,160,136,
136,160,136,136,136,0
10210 DATA 168,136,128,168,8,136,168,0,168,32,32,32,32,32,32,0,136,136,136,136,1
36,136,168,0
10220 DATA 136,136,136,136,136,136,32,0,136,136,136,136,168,168,136,0,136,136,13
6,32,32,136,136,0
10230 DATA 136,136,136,32,32,32,32,0,168,8,8,32,128,128,168,0
10240 DATA 16,84,68,68,84,68,68,0,80,68,68,80,68,68,80,0,84,68,64,64,64,68,84,0
10250 DATA 80,68,68,68,68,68,80,0,84,64,64,84,64,64,84,0,84,64,64,84,64,64,64,0
10260 DATA 84,68,64,69,68,68,84,0,68,68,68,84,68,68,68,0,84,16,16,16,16,16,84,0
10270 DATA 4,4,4,4,68,68,84,0,68,68,68,80,68,68,68,0,64,64,64,64,64,64,84,0
10280 DATA 68,84,84,68,68,68,68,0,68,68,84,84,84,84,68,0,84,68,68,68,68,68,84,0
10290 DATA 80,68,68,80,64,64,64,0,84,68,68,68,68,68,84,5,80,68,68,80,68,68,68,0
10300 DATA 84,68,64,84,4,68,84,0,84,16,16,16,16,16,16,0,68,68,68,68,68,68,84,0
10310 DATA 68,68,68,68,68,68,16,0,68,68,68,68,84,84,68,0,68,68,68,16,16,68,68,0
10320 DATA 68,68,68,16,16,16,16,0,84,4,4,16,64,64,84,0
```

Figure 1.

only at your own risk.

The 6502 microprocessor chip is the central processing unit of all current Atari machines, as it is for two of the "big three" machines. However in the Atari, the 6502 chip is backed up by three others, and therein lies a big difference.

One of these chips, called Antic, is itself a microprocessor. It is capable of an exotic potential known as "direct memory access," or DMA. Antic works in tandem with another chip, the GTIA or CTIA, to handle the video display, thus taking the weight of keeping the video screen "lit up" from the 6502. The CPU can go on to other important jobs.

I could attempt to outline each of the capabilities of these chips: 256 colors, up to 16 shades of a single color, 320 x 192 pixel resolution, player-missile graphics, modifiable display lists and character sets. However there really are only two ways to experience their power: watch an Atari graphics demo, or play a quality Atari game. The new GTIA chip, which replaces the CTIA, extends this power yet further.

Still another chip, called Pokey, generates, among other things, four channel sound. This sound can range from pure tone to many levels of distortion, allowing for music as well as sophisticated and complex sound effects generation. Sound is routed through the TV speaker, and so volume control is as simple as the flick of a knob. Sound can be routed just as simply to your stereo. Nearly all music composition and game playing in my home takes place through headphones.

The Atari is not without its problems. Much of the software written for it doesn't come close to truly utilizing its capabilities. It seems as if many programmers are having trouble realizing what power the Atari puts in their hands, and how best to use it. Dual density drives, 80-column capability, and truly professional word processing packages are only just now making an appearance.

But relief is in sight. It was a trickle at first, but third party software began to pour in. The trickle became a gush, and the gush became a torrent. Third party hardware followed soon after. The industry, realizing its initial underestimation of the machine, is compensating.

A variety of talented minds are working with the Atari, investing it with a variety of new capabilities. The machine offers one of the most exciting forefronts in the microcomputer industry today.

Incidentally, the big three will shortly have to move over. I predict by the end of this year Atari will be the number one microcomputer in its class, both in monthly sales and total units.

Multicolor Characters

Figure 1 is a short program with a very neat result: a multicolor character set in graphics 0. The idea goes back quite a ways: I remember first having seen it in *3-D Supergraphics,* from Paul Lutus. A recent example appears in the assembly language tutorial *Page Six,* from Synapse Software, which uses quite a well done font.

The technique involved in creating multicolor characters is called artifacting. This is the same phenomenon that sometimes causes ugly glitches in graphics 8 displays. By skipping adjacent pixels, red or blue characters can be formed, and artifacting can be used constructively.

The approach has its limitations. Because the default character size on the Atari is 8 x 8, skipping adjacent pixels results in a character three pixels wide. It is hard to create a font three pixels wide and at the same time keep N's and M's from looking alike, or support lower case.

In order to compensate, I made the font one scan line taller than the default value, and stuck to upper case. Still, I think you will agree the results are remarkable considering the constraints of the approach, and well worth taking the time to type.

Lines 10000 through 10020 define the point in memory at which we will start our redesigned character set. Lines 10030 and 10040 clear the screen, coloring it black, so that the artifacted character set will be clear. I suggest the altered set always be used on a black background.

Lines 10050 and 10060 set up the variables we will use to load the original character set into RAM, and later for overwriting the redefined characters. Line 10070 sets the character set pointer to the beginning of the RAM set. Line 10080 is placed there so you can watch the transformation take place; you can pull this line if you so desire.

Lines 10090 through 10110 load the entire original ROM character set into RAM. Then line 10120 replaces the upper case A through Z with values occurring up ahead as data statements. Likewise line 10130 replaces lower case a through z with newly defined character values.

The new upper case and lower case fonts are the same, with the exception of a one clock horizontal shift. This means that the upper case A through Z will be one color, and the lower a through z another. Because of differences in the way artifacting is handled by the GTIA as opposed to the CTIA, a GTIA machine will have, as a result of running this program, a blue upper case and red lower case, while a CTIA machine will have a red upper case and blue lower case. Not to mention what happens when printing inverse characters. Try it! All other characters, as well as numbers, will remain as default.

You might now incorporate this as a subroutine in other Basic programs (remember to stick a RETURN in there somewhere, and keep the program from hitting line 10000 other than through that initial GOSUB).

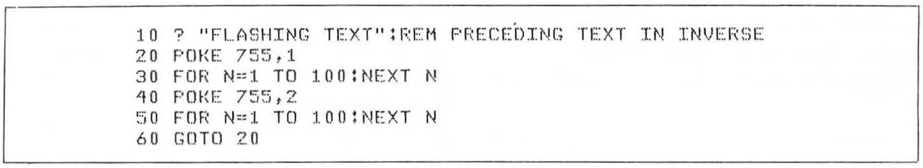
Sheldon Leemon, on whose program, *Instedit,* I designed the font, reminded me that the display list could be modified to display the fonts in any color. I may take up this challenge in a subsequent Outpost. For now, I will leave it to you. List the program in the modified set; you will see that it can even function as a programming tool.

Poking Around

As a result of my comments about memory locations in the November column, I got a slightly indignant letter from Becky Johnson, at Educational Software (formally Santa Cruz Educational Software). She reminded me that their publication *Master Memory Map* had sold more than 10,000 copies at $6.95. Well I admit I hadn't seen the publication at the time, and though it is still not a truly *definitive* list, it has got to be the closest yet. If you wish more information, you can contact them at 4564 Cherryvale Ave., Soquel, CA 95073. (408) 476-4901.

In the meantime, here are some more interesting locations to keep you busy:

Disabling the break key. POKE 16,64



```
10 ? "FLASHING TEXT":REM PRECEDING TEXT IN INVERSE
20 POKE 755,1
30 FOR N=1 TO 100:NEXT N
40 POKE 755,2
50 FOR N=1 TO 100:NEXT N
60 GOTO 20
```

Figure 2.

and POKE 53774,64 to disable the break key. Very handy to keep users from interrupting or getting into a program.

Disabling DMA. PEEK (559), then POKE 559,0. This will shut down Antic, allowing the 6502 to speed execution dramatically. POKE 559 with value initially PEEKed to re-enable screen display. Also handy as a "curtain," in concealing the screen during display initialization or other potentially distracting moment. This is as opposed, for example, to resetting graphics mode and setting color registers to black.

Putting a text window into graphics 0. POKE 703,4. This will force all normal text into a text window as in graphics modes. Printing to the upper part of the screen must be accomplished with PRINT #6 statements. Could be handy in writing text adventures (maybe even with the multicolor font). To return to default, POKE 703,24.

Flashing characters. Set up a loop wherein the value of location 755 varies from the normal, 2, to 1. Figure 2 is an approach to flashing characters.

It is a nice attention getter in programs. We will also look at more sophisticated means of obtaining flashing characters in an upcoming column.

Checking for keypresses. POKE 764,255, then PEEK(764) for internal keycode. Handy to check for any or a specific keypress. Can also be used to "press a key" through software: for example, POKE 764,12 will RETURN automatically.

To enable cassette recorder. POKE 54018,52 to turn cassette play on, POKE 54018,60 to turn it off. Recorder must, of course, be set with cassette in place and play key pressed. Use to sync recorded sound with programs. □

Self-Modifying Programs

John Anderson

Original Atari Basic has its strong and weak points, as do all computer languages. Because Atari Basic is a somewhat renegade dialect, however (as opposed to the orthodoxy of Microsoft Basic), it is subject to especially intense scrutiny. Those who dislike it tend to detest it; even those who like it tend toward ambivalence. C'est la langue.

There is at least one good reason why Atari Basic is a "splinter language." It was designed in tandem with and in order to squeeze the most from the Atari operating system. And as such, it is capable of some exotic tricks—that much is undeniable.

One of these tricks is the ability to write-code that in turn rewrites itself. Imagine the possibilities.

The Atari has a very open-minded operating system. It will allow the screen editor to operate from sources other than a human at the keyboard. The editor will go so far as to accept data pushed to the screen from Atari Basic, and Atari Basic can then execute commands directly from the screen editor.

Figure 1 is a short example of how this feature can work. First we clear the screen, and position the cursor. Then we straightforwardly print code lines to the screen. These lines will act as if the Atari has automatically pushed the RETURN key over them, thus incorporating them into the program (and eliminating any previous lines with the same line numbers). Notice the inclusion of a CONTinue command. You must print this command at the bottom of any list of modifications or the program will terminate before modification takes place. The program run must actually stop, accept the new data, and start itself again.

The trickiest facet of the technique is placement of the cursor. If you position it incorrectly, you can lose modifications, or get locked into a loop. A bit of experimentation will lead to successful results.

Only now is the potential of this capability being fully explored. Two new programs from Artworx Software make use of the technique: *Drawpic* saves four color user drawings in graphics modes 3 through 7, by saving modified strings; *Player/Missile Editor* does the same for player/missile shape tables. You can get more

John Anderson is an associate editor for *Creative Computing* magazine.

information concerning these programs from Artworx, 150 North Main Street, Fairport, NY 14450.

A hint on how you might simply utilize the technique in your own programs is shown in Figure 2. Here the user is asked to input data, which is then incorporated into the modified program. This program can then be saved, thus saving the input information.

I've used the simplest possible approach in this example, saving up to one hundred phone numbers as REM statements. You might wish to improve radically on this approach.

Another use of the technique would be to delete lines when they are no longer needed. Line numbers devoted to user input of variables, for example, could be deleted after the variable table has been constructed. This would help conserve memory.

Countless other applications await *your* entry into Atari behavior modification. The limits are set by your imagination only.

Souping Your Machine

Although the Atari does most things well, you can now customize it to do things better. The idea may seem to you akin to putting slicks on a BMW, but let me tell you about a few products we've tested that can make your machine faster and more versatile.

The *Fastchip*, from Newell Industries, replaces the floating point chip on the operating system board. Floating point routines, which involve mathematical operations with real numbers as well as integers, run extremely slowly on a standard Atari. Newell Industries claims that execution of these routines is boosted to three times the original rate.

I played *Hail To the Chief* twice on the same machine, once with the original chip, and once with *Fastchip*. Calculations within the program involve lengthy breaks in the action. *Fastchip* cut the waiting at least in half, from a maximum of 11 seconds to a maximum of about 5 seconds. It may not sound like much of a difference, but when you're waiting it is.

If you are into floating point routines and don't have a lot of time, *Fastchip* will help. It lists for $39.95, and installation in an Atari *800* takes less than five minutes. For more information, contact Newell Industries, 3340 Nottingham Lane, Plano, TX 75074.

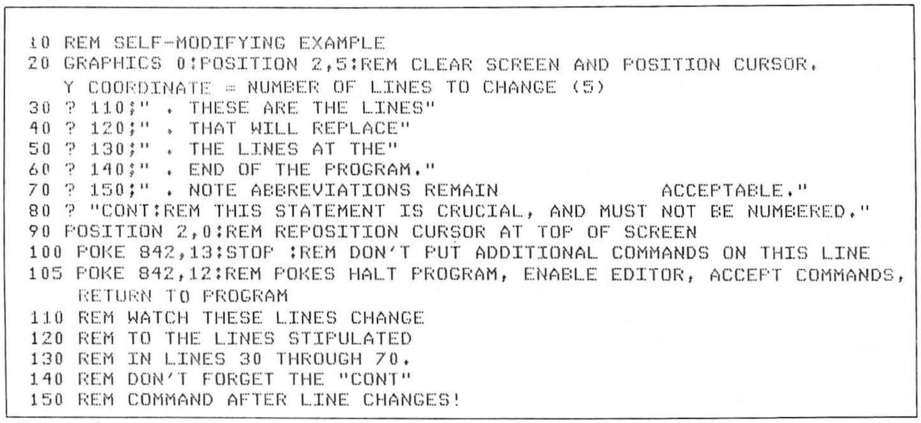
You should resist with all your strength the temptation to confuse *Fastchip* with *Fast Chip*, the disk drive upgrade chip from Binary Corporation. This product will interest Atari owners with original 810 disk drives, as it provides a disk format 30% faster than the original. The company claims that the custom chip is 10% faster than even the new Atari upgrade chip.

It took about a half an hour for me to perform the upgrade, and was a bit more involved than I had initially anticipated. Still, the instructions are clear, and the process is broken into logical steps.

Binary Fast Chip without a doubt provides a faster format for your disks. I found that it cut about twenty seconds off the load time of a 12K file. But the disks are also rather sensitive —cases arose wherein *Fast Chip*-formatted disks took much *longer* to read or write. This was without exception true with disks for use with *Valforth*. It might therefore be advisable to wait until you have two drives, then install a *Fast Chip* in one. You can then choose the format to match the application.

The product lists for $39.95. For more information, contact Binary Corporation, 3237 Woodward Ave., Berkley, MI 48072.

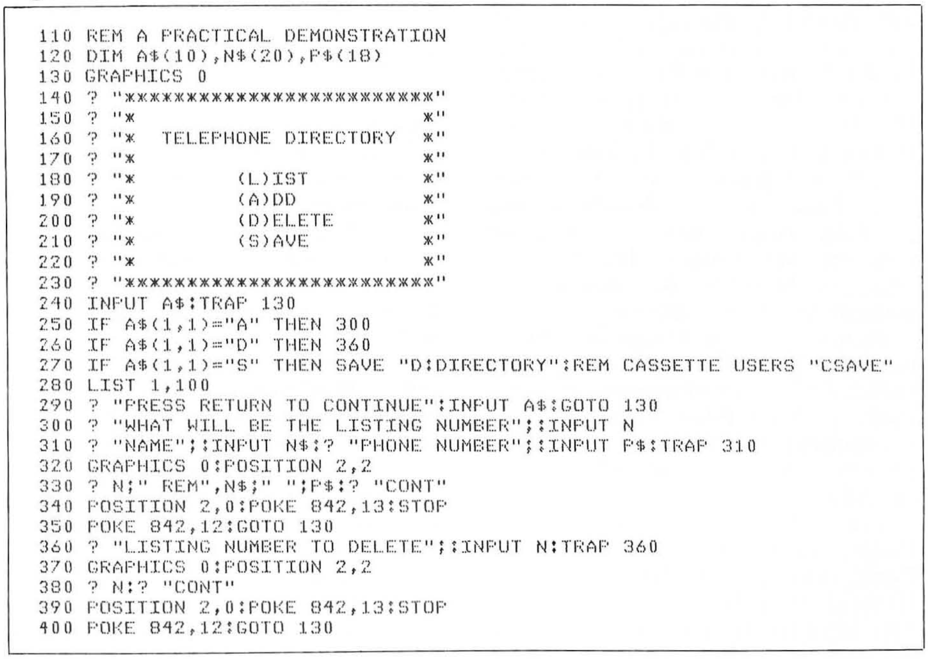
Perhaps you've wondered if ROM cartridges could be copied to disk. Well they can now, with the *Block* from Protronics. The *Block* allows you to transfer from a ROM to a binary disk file. Up to ten ROM cartridges can be saved to a single disk. I was unable to find anything that the *Block* couldn't copy.



```
10 REM SELF-MODIFYING EXAMPLE
20 GRAPHICS 0:POSITION 2,5:REM CLEAR SCREEN AND POSITION CURSOR.
   Y COORDINATE = NUMBER OF LINES TO CHANGE (5)
30 ? 110;" . THESE ARE THE LINES"
40 ? 120;" . THAT WILL REPLACE"
50 ? 130;" . THE LINES AT THE"
60 ? 140;" . END OF THE PROGRAM."
70 ? 150;" . NOTE ABBREVIATIONS REMAIN             ACCEPTABLE."
80 ? "CONT:REM THIS STATEMENT IS CRUCIAL, AND MUST NOT BE NUMBERED."
90 POSITION 2,0:REM REPOSITION CURSOR AT TOP OF SCREEN
100 POKE 842,13:STOP :REM DON'T PUT ADDITIONAL COMMANDS ON THIS LINE
105 POKE 842,12:REM POKES HALT PROGRAM, ENABLE EDITOR, ACCEPT COMMANDS,
    RETURN TO PROGRAM
110 REM WATCH THESE LINES CHANGE
120 REM TO THE LINES STIPULATED
130 REM IN LINES 30 THROUGH 70.
140 REM DON'T FORGET THE "CONT"
150 REM COMMAND AFTER LINE CHANGES!
```

Figure 1.

Figure 2.



```
110 REM A PRACTICAL DEMONSTRATION
120 DIM A$(10),N$(20),P$(18)
130 GRAPHICS 0
140 ? "*************************"
150 ? "*                       *"
160 ? "*  TELEPHONE DIRECTORY  *"
170 ? "*                       *"
180 ? "*        (L)IST         *"
190 ? "*        (A)DD          *"
200 ? "*        (D)ELETE       *"
210 ? "*        (S)AVE         *"
220 ? "*                       *"
230 ? "*************************"
240 INPUT A$:TRAP 130
250 IF A$(1,1)="A" THEN 300
260 IF A$(1,1)="D" THEN 360
270 IF A$(1,1)="S" THEN SAVE "D:DIRECTORY":REM CASSETTE USERS "CSAVE"
280 LIST 1,100
290 ? "PRESS RETURN TO CONTINUE":INPUT A$:GOTO 130
300 ? "WHAT WILL BE THE LISTING NUMBER";:INPUT N
310 ? "NAME";:INPUT N$:? "PHONE NUMBER";:INPUT P$:TRAP 310
320 GRAPHICS 0:POSITION 2,2
330 ? N;" REM",N$;" ";P$:? "CONT"
340 POSITION 2,0:POKE 842,13:STOP
350 POKE 842,12:GOTO 130
360 ? "LISTING NUMBER TO DELETE";:INPUT N:TRAP 360
370 GRAPHICS 0:POSITION 2,2
380 ? N:? "CONT"
390 POSITION 2,0:POKE 842,13:STOP
400 POKE 842,12:GOTO 130
```

Potential pirates should take note: the *Block* itself is a ROM cartridge, and no cartridge file will run unless the *Block* is installed.

The *Block* lists for $99.95. For more information, contact Protronics, 17537 Chatsworth, Granada Hills, CA 91344.

The Library Grows

I remember, in the dim recesses of my mind, a time when information about the Atari was an extremely rare commodity. This was in ancient times, of course: maybe a year and a half ago. Now it seems a new book about the Atari arrives here every week. These are six of the best:

Atari Games and Recreations, by Herb Kohl, Ted Kahn, Len Lindsay, and Pat Cleland, Reston Publishing, Reston, VA 22090. Excellent starter for novices and kids, with an emphasis on fun programs the user can type in, play, and understand. Includes some nifty appendices.

Atari Sound and Graphics, by Herb Moore, Judy Lower, and Bob Albrecht, John Wiley and Sons, 605 Third Avenue, New York, NY 10158. The authors pace the text so that new concepts are introduced at a rate that can be absorbed. Sound and graphics are a motivating force with kids, but many hobbyists will want this one too.

The Atari Assembler, by Don Inman and Kurt Inman, Reston Publishing, Reston, VA 22090. Best beginners machine language book available for Atari owners. Assumes you have Basic and an editor/assembler. Assembly language is tough stuff, but authors manage to keep things fresh with humor and good examples.

Games for the Atari, by S. Roberts, W. Hofacker, 53 Redrock Lane, Pomona, CA 91766. Includes good examples of player/missile movement from Basic, priority detection, and patching from Basic to machine language subroutines. Includes ten games to be typed in, unfortunately without much explanation.

Picture This, by David D. Thornburg, Addison-Wesley, Reading, MA 01867. A kid's introduction to graphics through Atari Pilot. Excellent as a supplement to "Student Pilot," the reference guide supplied with the Pilot cartridge.

Your Atari Computer, by Lon Poole, Martin McNiff, and Steven Cook, Osborne McGraw-Hill, 630 Bancroft Way, Berkeley, CA 94710. It may have taken two and a half years, but there is finally a manual available which thoroughly documents the rudiments, as well as a number of advanced topics, concerning Atari personal computers.

Following a remarkably steady pace, the book progresses through beginning operation, getting started in Basic programming, and includes comprehensive chapters on the program recorder, disk drive, and printers.

The main body of the book deals with advanced Basic programming, and stands to serve the proficient Basic programmer as well as the novice.

Next the book focuses on the goodies. Graphics and sound are given a clear and thorough treatment — with a chapter devoted to advanced graphics techniques. Character set animation, display lists and player-missile graphics are explained simply and thoroughly, with examples helping to illuminate the way. Those of you (like myself) who need every last thing spelled out for you, are bound to benefit from this approach.

Later chapters examine sound routines and summarize Basic commands.

The book concludes with nine appendices, each of value to the Atari programmer. Finally we can turn to a single resource for an annotated list of error codes and their meanings, status and keyboard codes, memory usage charts, and a listing of important memory locations.

Your Atari Computer should be packed with each and every unit Atari ships, alongside or in lieu of current documentation. No Atari owner should be without it.

Scuttlebytes

In the November issue, we gave a phone number for a Sunnyvale bulletin board system called TEAM Atari. Begun by an Atari employee, the board is unfortunately no longer in service. Apparently some Atari execs felt it was inappropriate, which is too bad.

You might try Bay Area Atari at (408) 244-6229. Sorry if we caused any inconvenience, but it is tough to compile a BBS list that remains totally accurate for any length of time.

A question many people are asking concerns the new Atari *5200* video system: is it or isn't it a *400* without a keyboard? The answer: well, yes and no. It does have 6502, Antic, GTIA, and Pokey chips. It does run nearly identical ROM software. However, for reasons somewhat difficult to fathom, the *5200* has had enough changes made to ensure incompatibility with Atari computers. The most dramatic evidence of this is a redesigned game controller, which uses an analog input, in addition to a telephone-style keypad.

The advantages to a handheld keypad are obvious: the advantages of an analog joystick perhaps less so. A potentiometer-controlled stick allows for better control in games such as *Missile Command,* but a digital stick is faster in quick-turning games like *Pac-Man.* The *5200* controller ports are necessarily redesigned, as are the cartridges themselves. Whether this incompatibility is utterly surmountible remains to be seen, but it certainly would be a formidable task.

Another topic we hear a lot about is the "next generation" Atari. Have you heard about the Atari *600?* We have, although we haven't been able to confirm anything. It will be a single board computer, totally compatible with the *400* and *800.* It will have RS-232 capability built-in, and a full-stroke keyboard. It will come with 48K standard, and sport programmable function keys. We have even heard about an Atari *1000,* with a built-in dual density drive, and CP/M capability!

Super Text Mode

John Anderson

After a program is written and debugged, it should be cleaned and polished. When you think of all the work you did to get it working, the least you can do is mount it well—and that entails making it look and run right.

The first display a program generates is supremely important, as it sets the tone for all that is to follow. If you have been wanting a professional quality title card to distinguish your programs, here is a routine that will fill the bill. It can be tailored to display your message in a large, custom font, and then to cycle through a veritable rainbow of color. From there, another message can flash into the text window. After the title card has cycled fully, the rest of your program will execute.

Programs that plot and fill character sets into a non-text mode (in this case graphics 7) have appeared in *Byte,* and *Compute!* over the past two years. The routine that follows has some new features, and allows you to create your own, customized displays with a minimum of fuss.

If you are short on memory for a specific application, or don't have a disk drive, you might not want to commit many lines of Basic to the likes of this routine. However if you have a disk, the program can run as a separate file, invoking your main program file as its final act. It takes about 50 sectors on disk, and is well worth every bit of that space.

The routine appears in Listing 1. I have left the code relatively free of REM statements to conserve memory. If typed in exactly as shown, the program will display the letters A through U on the screen, cycle through the rainbow and text window displays, and then start all over. In order to display letters V through Z, change line 120 to RESTORE 862. This will at least give you a view of all the letters in the font, so you can ensure that the program has been entered correctly. The only reason the alphabet has been split in this manner is because the screen is capable of displaying only three rows of seven characters at one time.

The program breaks down as follows:

Line 10 initializes, while POKE statements suppress the cursor and move the margins out to 40 columns. It also invokes two subroutines: the first reads machine language data into a string which will then execute the rainbow segment from a USR call in line 30. The second subroutine reads data which define the placement and choice of characters.

Lines 50 through 70 constitute the secondary message, which will appear in the text window. There is no reason why this font, which is in graphics 0, could not be modified, perhaps along the lines of the program that appeared in the *Upstart Atari* article (see pages 135-138).

Line 80 creates a pregnant pause, then starts the whole procedure over again. Before it does so, however, it POKES the attract mode into operation. You may or may not like the effect this creates, but the command shows that color background capability is there. This is also the line from which the rest of another program would take off.

Lines 130 through 230 are really the heart of the program, and exemplify a powerful and efficient manner of reading plotting (as well as other) information from an upcoming series of DATA statements. This handsome approach has appeared in *Compute!*, though I have improved upon it here.

Briefly, the following happens when reading DATA statements: if preceded by a P, upcoming data pertains to plotting. Read the numbers, PLOT and DRAWTO as necessary.

If preceded by an R, the following data will indicate where the plot should begin (always at the 0,0 point of any letter). These numbers always occur at the outset of a letter plot, and must be manipulated by the user in order to ensure correct placement. The font is "proportionally spaced"; for example, an I is narrower than an M, and care must be taken to lay out words so that spacing between letters is pleasing. Shades of art direction! The first number is the vertical coordinate, the second the horizontal. And remember, the Atari does *not* use the Cartesian coordinate system, but rather places 0,0 at the upper left-hand corner.

If preceded by an S, the data pertain to sound statements. Thus the user can create a tone for each letter, building into and moving between chords, if so desired, by cycling voices. The first number indicates which voice to use, the second what pitch. I worked exclusively with pure tone (10). By altering the distortion value in line 160, you can experiment with sound effects.

The next data identifier may seem a bit mysterious, as it is not used in the demonstration version of the program. It is a dummy identifier, placed there only for possible use as a time delay. As certain letters require fewer steps to draw than others (I as opposed to S, for example, they will plot much more quickly. By padding the DATA statements for the letter I with D's, you would be able to even out its plot time, to create a truly professional-looking display.

If preceded by a letter F, the numbers indicate a following fill statement. These ensure that the insides of each character will be delineated from the outside, so that the rainbow can then well up from within. And if the word END is encountered, the job of this section of code is terminated.

What follows are the somewhat lengthy DATA statements themselves. "Why the jump from line number 230 to 650?" the more observant Atarians out there may ask. "Well, for a good reason," I respond. The line numbers that initiate each letter correspond exactly to their ATASCI code times 10.

An observant but somewhat slower subgroup might ask as a follow-up, "so what?" Well I'm glad you asked that question. In a subsequent version of this program, (which I haven't yet written because I am hoping one of you will do it for me), the user will be able to input an entire message into a character string, and through the use of techniques outlined in *Self-Modifying Programs* (pages 138-140), the program will then *modify* itself into the specific message, deleting all extraneous material.

A hint at a possible approach: 1) find out what letter the user wants, 2) LIST CHR$ (user's letter) *10+2, give it a new line number (how about that gap between lines 230 and 650), then re-enter the line. Do this for +4, +6, and +8, and you will have the entire letter reentered. Preset R (origin) and S (sound) values that the user can fiddle with later. There will be plenty of time for that, what with the time saved not having to edit the whole thing by hand. Then have the program automatically delete lines 650 through 904 completely. Voila, a custom title card, in minimal memory!

Disk Utilities

Want to learn more about how your disk drives work? Need a way to retrieve data from crashed disks? Want to look at and alter disk information sector by sector? Interested in backing up your disks? If your answer to any of these questions is yes, you are a candidate for a disk utility package.

Every time I have been about ready to write something on disk utilities, another package makes makes its appearance. The latest I have had a chance to become acquainted with is *Diskey* from Adventure International. This packs the most features I have yet seen in a disk

John Anderson is an associate editor for *Creative Computing* magazine.

utility, and is accompanied by a rare bonus: sparkling, well-written documentation by the software author himself. The tutorial value of the manual alone makes the package worthwhile.

But just wait until you see the software. Disk maps are presented simultaneously in hexadecimal and ASCII format. The software allows for sector-by-sector data examination and alteration. It provides tools by which to salvage damaged disks. It provides functions to compare, copy, reformat, search, create disk files from tape autoboot files, erase (write zeros), disable verify, calibrate drive speed, and manipulate disk directories as well as DOS files.

Another unique function of *Diskey* is its ability to flag "dead" disk sectors. This is a capability previously unavailable in any disk utility I have seen. Though the potential for misuse is there, author Sparky Starks stops short of spelling out a means to write bad sectors. He states strongly in the manual foreword his equation of software pirates with common thieves.

As a learning tool, *Diskey* is superlative. The documentation and software work in tandem to provide the most solid disk tutorial you can find anywhere. And, hard as it may seem to imagine, even the driest stuff is presented in a fresh, almost breezy manner.

Diskey is more than a professional utility: it is obviously a labor of love. There are many more features in the software than would have been necessary to create a salable package. For over 50 reasons (the product has over 50 separate commands), *Diskey* very quickly attained a pre-eminent position in my utilities box. How about an assembly language tutorial, Mr. Starks?

Obviously, you must have a disk drive to run the package; in addition, you must have at least 32K and Atari Basic. The system is optimally configured, however, with a 48K system, two drives, and an 80-column printer like the Atari 825.

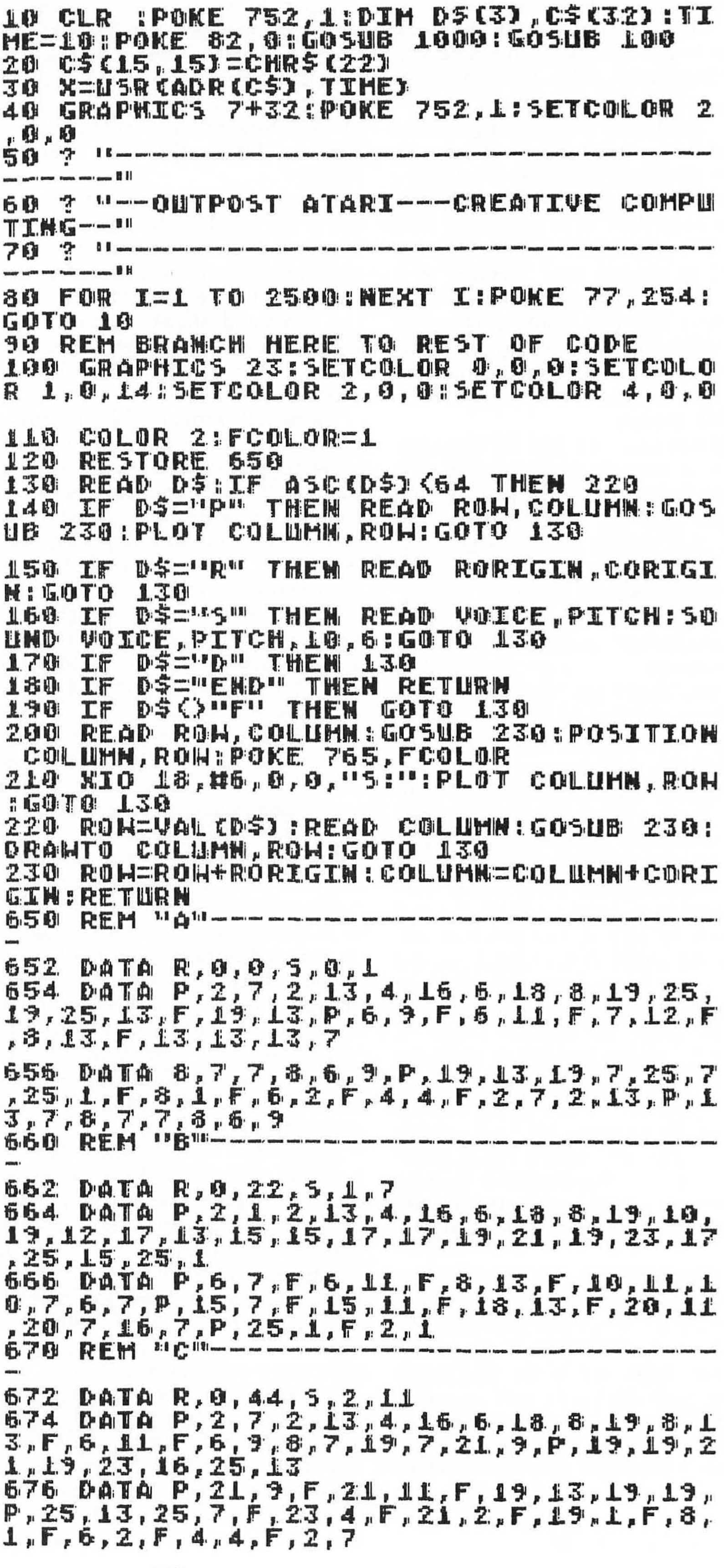
The package lists for $49.95. For more information, contact Adventure International, Box 3435, Longwood Fl., 32750. (305) 830-8194.

Poking Around

This part of the column was initiated as an attempt to respond to the many questions we have received at the magazine concerning Atari memory map locations, and ways of "tweaking" them.

I have at least three letters from Atari Basic hobbyists, all asking the same

Listing 1.



```
10 CLR :POKE 752,1:DIM D$(3),C$(32):TI
ME=10:POKE 82,0:GOSUB 1000:GOSUB 100
20 C$(15,15)=CHR$(22)
30 X=USR(ADR(C$),TIME)
40 GRAPHICS 7+32:POKE 752,1:SETCOLOR 2
,0,0
50 ? "--------------------------------
-------"
60 ? "--OUTPOST ATARI---CREATIVE COMPU
TING--"
70 ? "--------------------------------
-------"
80 FOR I=1 TO 2500:NEXT I:POKE 77,254:
GOTO 10
90 REM BRANCH HERE TO REST OF CODE
100 GRAPHICS 23:SETCOLOR 0,0,0:SETCOLO
R 1,0,14:SETCOLOR 2,0,0:SETCOLOR 4,0,0

110 COLOR 2:FCOLOR=1
120 RESTORE 650
130 READ D$:IF ASC(D$)<64 THEN 220
140 IF D$="P" THEN READ ROW,COLUMN:GOS
UB 230:PLOT COLUMN,ROW:GOTO 130

150 IF D$="R" THEN READ RORIGIN,CORIGI
N:GOTO 130
160 IF D$="S" THEN READ VOICE,PITCH:SO
UND VOICE,PITCH,10,6:GOTO 130
170 IF D$="D" THEN 130
180 IF D$="END" THEN RETURN
190 IF D$<>"F" THEN GOTO 130
200 READ ROW,COLUMN:GOSUB 230:POSITION
 COLUMN,ROW:POKE 765,FCOLOR
210 XIO 18,#6,0,0,"S:":PLOT COLUMN,ROW
:GOTO 130
220 ROW=VAL(D$):READ COLUMN:GOSUB 230:
DRAWTO COLUMN,ROW:GOTO 130
230 ROW=ROW+RORIGIN:COLUMN=COLUMN+CORI
GIN:RETURN
650 REM "A"---------------------------
-

652 DATA R,0,0,S,0,1
654 DATA P,2,7,2,13,4,16,6,18,8,19,25,
19,25,13,F,19,13,P,6,9,F,6,11,F,7,12,F
,8,13,F,13,13,13,7

656 DATA 8,7,7,8,6,9,P,19,13,19,7,25,7
,25,1,F,8,1,F,6,2,F,4,4,F,2,7,2,13,P,1
3,7,8,7,7,8,6,9
660 REM "B"---------------------------
-

662 DATA R,0,22,S,1,7
664 DATA P,2,1,2,13,4,16,6,18,8,19,10,
19,12,17,13,15,15,17,17,19,21,19,23,17
,25,15,25,1
666 DATA P,6,7,F,6,11,F,8,13,F,10,11,1
0,7,6,7,P,15,7,F,15,11,F,18,13,F,20,11
,20,7,16,7,P,25,1,F,2,1
670 REM "C"---------------------------
-

672 DATA R,0,44,S,2,11
674 DATA P,2,7,2,13,4,16,6,18,8,19,8,1
3,F,6,11,F,6,9,8,7,19,7,21,9,P,19,19,2
1,19,23,16,25,13
676 DATA P,21,9,F,21,11,F,19,13,19,19,
P,25,13,25,7,F,23,4,F,21,2,F,19,1,F,8,
1,F,6,2,F,4,4,F,2,7
```

question: how can Basic programs be made unlistable? First of all, let me go on record as one of the category of folks who believe in keeping things listable wherever and whenever it is feasible to do so. The problem of code theft is not nearly as acute in Basic as it is in machine language, nor is that diehard pirate going to be deterred by the mere fact that a program is unlistable in its usual environment. My feeling is, in the spirit of enlightenment, if other people stand to learn something from a bit of my code, more power to them.

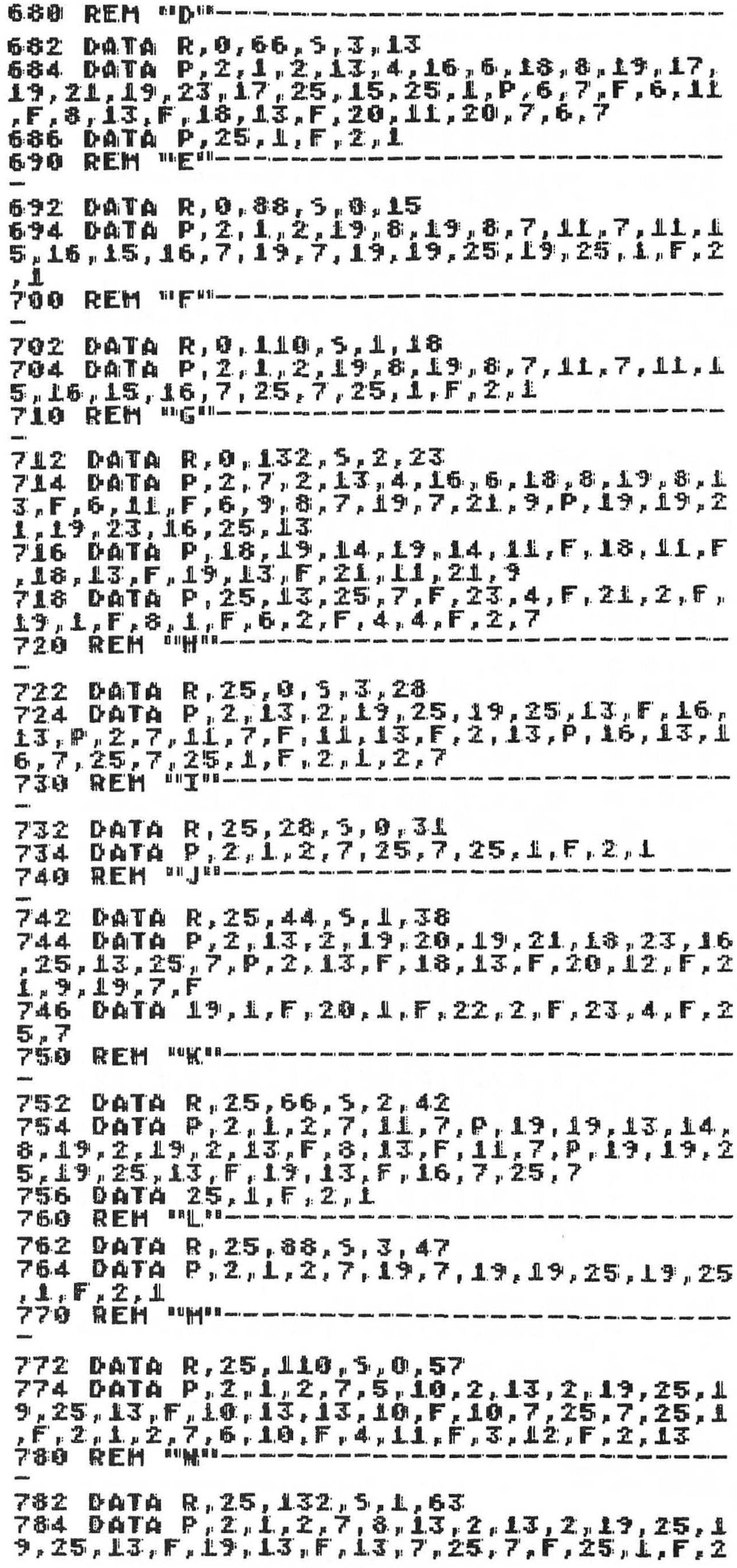
That disclaimer having been duly filed, let's look at the only tried, true, and simple method I have seen to help protect your precious Basic files from prying eyes.

Most approaches I have seen to rendering Basic programs unlistable are unsatisfactory. In my *Upstart Atari* article, I noted the memory locations you can alter to disable the BREAK key (see **Poking Around,** pp. 137-138). (A quick aside—a couple of folks wrote in telling me they experienced problems disabling BREAK. The POKE commands must be reasserted often. For instance, POKE again after every graphics mode command. If you put enough sets of them in your program or stick them in the right places in the main loops, you will effectively disable the key.)

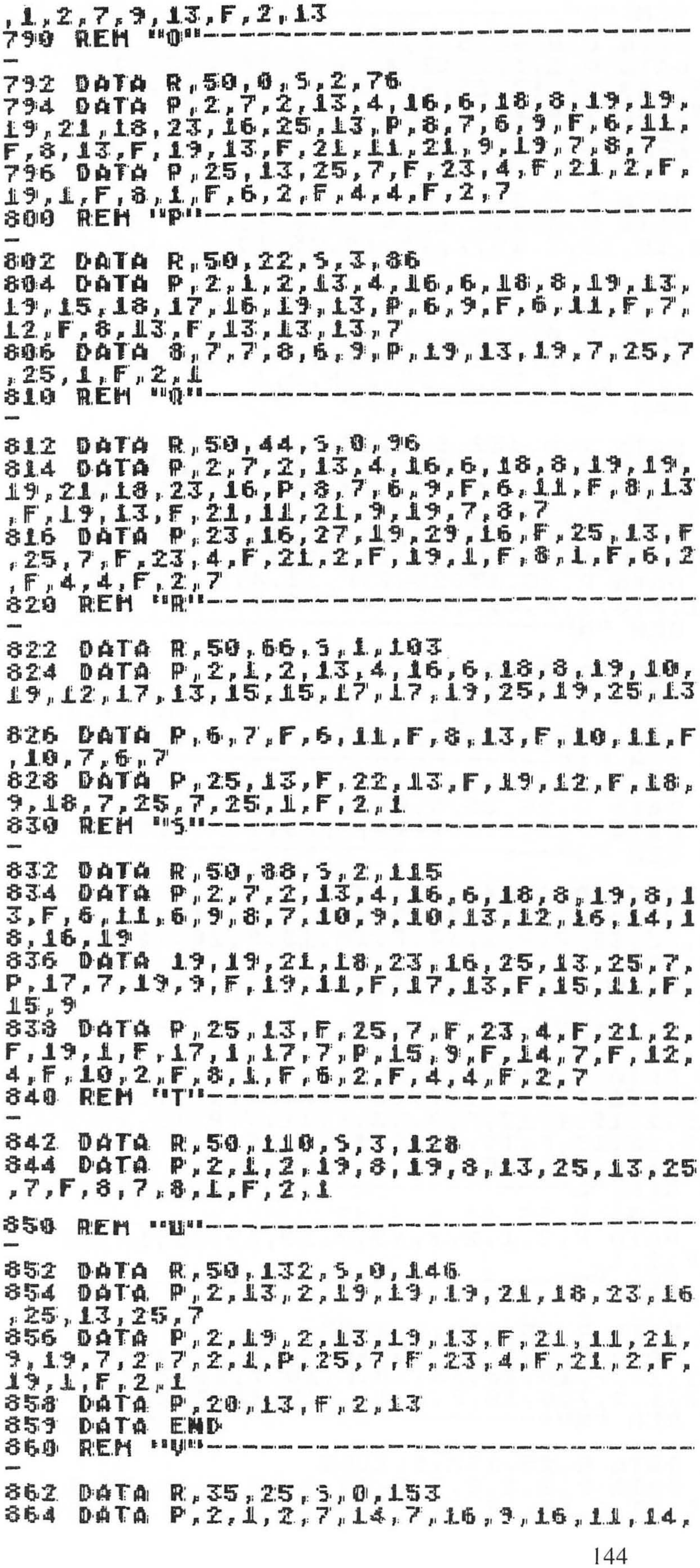
I have not found a way to disable the RESET key, but with the command POKE 580,1 you can make the key into a "true" reset: that is, pressing the key will initiate a cold start, as if the system has been powered down and up again. This will flush any resident program from memory. To return to the normal RESET mode, POKE 581, 1.

The trouble with merely disabling these keys is that the program can still be listed before it is ever run. Still another approach I have seen converts program listings into control characters or variables into carriage returns. Likewise, the fixes do not become operative until the programs are run. If the user asks for a LIST directly after loading, a full listing will be obtained.

How then, to protect a program before it can be listed? The answer lies in the creation of a "RUN only" file. This type of file can not be LOADed or ENTERed, nor can it ever be LISTed. It executes perfectly in every other respect, but can only be invoked with the command RUN"D:FILENAME", (or RUN "C:" if you are using a cassette-based system). In order to create such a file, append the following line:



```
680 REM "D"---------------------------
682 DATA R,0,66,5,3,13
684 DATA P,2,1,2,13,4,16,6,18,8,19,17,
19,21,19,23,17,25,15,25,1,P,6,7,F,6,11
,F,8,13,F,18,13,F,20,11,20,7,6,7
686 DATA P,25,1,F,2,1
690 REM "E"---------------------------
-
692 DATA R,0,88,5,0,15
694 DATA P,2,1,2,19,8,19,8,7,11,7,11,1
5,16,15,16,7,19,7,19,19,25,19,25,1,F,2
,1
700 REM "F"---------------------------
-
702 DATA R,0,110,5,1,18
704 DATA P,2,1,2,19,8,19,8,7,11,7,11,1
5,16,15,16,7,25,7,25,1,F,2,1
710 REM "G"---------------------------
-
712 DATA R,0,132,5,2,23
714 DATA P,2,7,2,13,4,16,6,18,8,19,8,1
3,F,6,11,F,6,9,8,7,19,7,21,9,P,19,19,2
1,19,23,16,25,13
716 DATA P,18,19,14,19,14,11,F,18,11,F
,18,13,F,19,13,F,21,11,21,9
718 DATA P,25,13,25,7,F,23,4,F,21,2,F,
19,1,F,8,1,F,6,2,F,4,4,F,2,7
720 REM "H"---------------------------
-
722 DATA R,25,0,5,3,28
724 DATA P,2,13,2,19,25,19,25,13,F,16,
13,P,2,7,11,7,F,11,13,F,2,13,P,16,13,1
6,7,25,7,25,1,F,2,1,2,7
730 REM "I"---------------------------
-
732 DATA R,25,28,5,0,31
734 DATA P,2,1,2,7,25,7,25,1,F,2,1
740 REM "J"---------------------------
-
742 DATA R,25,44,5,1,38
744 DATA P,2,13,2,19,20,19,21,18,23,16
,25,13,25,7,P,2,13,F,18,13,F,20,12,F,2
1,9,19,7,F
746 DATA 19,1,F,20,1,F,22,2,F,23,4,F,2
5,7
750 REM "K"---------------------------
-
752 DATA R,25,66,5,2,42
754 DATA P,2,1,2,7,11,7,P,19,19,13,14,
8,19,2,19,2,13,F,8,13,F,11,7,P,19,19,2
5,19,25,13,F,19,13,F,16,7,25,7
756 DATA 25,1,F,2,1
760 REM "L"---------------------------
762 DATA R,25,88,5,3,47
764 DATA P,2,1,2,7,19,7,19,19,25,19,25
,1,F,2,1
770 REM "M"---------------------------
-
772 DATA R,25,110,5,0,57
774 DATA P,2,1,2,7,5,10,2,13,2,19,25,1
9,25,13,F,10,13,13,10,F,10,7,25,7,25,1
,F,2,1,2,7,6,10,F,4,11,F,3,12,F,2,13
780 REM "N"---------------------------
-
782 DATA R,25,132,5,1,63
784 DATA P,2,1,2,7,8,13,2,13,2,19,25,1
9,25,13,F,19,13,F,13,7,25,7,F,25,1,F,2
```



```
,1,2,7,9,13,F,2,13
790 REM "O"------------------------------
-
792 DATA R,50,0,5,2,76
794 DATA P,2,7,2,13,4,16,6,18,8,19,19,
19,21,18,23,16,25,13,P,8,7,6,9,F,6,11,
F,8,13,F,19,13,F,21,11,21,9,19,7,8,7
796 DATA P,25,13,25,7,F,23,4,F,21,2,F,
19,1,F,8,1,F,6,2,F,4,4,F,2,7
800 REM "P"------------------------------
-
802 DATA R,50,22,5,3,86
804 DATA P,2,1,2,13,4,16,6,18,8,19,13,
19,15,18,17,16,19,13,P,6,9,F,6,11,F,7,
12,F,8,13,F,13,13,13,7
806 DATA 8,7,7,8,6,9,P,19,13,19,7,25,7
,25,1,F,2,1
810 REM "Q"------------------------------
-
812 DATA R,50,44,5,0,96
814 DATA P,2,7,2,13,4,16,6,18,8,19,19,
19,21,18,23,16,P,8,7,6,9,F,6,11,F,8,13
,F,19,13,F,21,11,21,9,19,7,8,7
816 DATA P,23,16,27,19,29,16,F,25,13,F
,25,7,F,23,4,F,21,2,F,19,1,F,8,1,F,6,2
,F,4,4,F,2,7
820 REM "R"------------------------------
-
822 DATA R,50,66,5,1,103
824 DATA P,2,1,2,13,4,16,6,18,8,19,10,
19,12,17,13,15,15,17,17,19,25,19,25,13

826 DATA P,6,7,F,6,11,F,8,13,F,10,11,F
,10,7,6,7
828 DATA P,25,13,F,22,13,F,19,12,F,18,
9,18,7,25,7,25,1,F,2,1
830 REM "S"------------------------------
-
832 DATA R,50,88,5,2,115
834 DATA P,2,7,2,13,4,16,6,18,8,19,8,1
3,F,6,11,6,9,8,7,10,9,10,13,12,16,14,1
8,16,19
836 DATA 19,19,21,18,23,16,25,13,25,7,
P,17,7,19,9,F,19,11,F,17,13,F,15,11,F,
15,9
838 DATA P,25,13,F,25,7,F,23,4,F,21,2,
F,19,1,F,17,1,17,7,P,15,9,F,14,7,F,12,
4,F,10,2,F,8,1,F,6,2,F,4,4,F,2,7
840 REM "T"------------------------------
-
842 DATA R,50,110,5,3,128
844 DATA P,2,1,2,19,8,19,8,13,25,13,25
,7,F,8,7,8,1,F,2,1

850 REM "U"------------------------------
-
852 DATA R,50,132,5,0,146
854 DATA P,2,13,2,19,19,19,21,18,23,16
,25,13,25,7
856 DATA P,2,19,2,13,19,13,F,21,11,21,
9,19,7,2,7,2,1,P,25,7,F,23,4,F,21,2,F,
19,1,F,2,1
858 DATA P,20,13,F,2,13
859 DATA END
860 REM "V"------------------------------
-
862 DATA R,35,25,5,0,153
864 DATA P,2,1,2,7,14,7,16,9,16,11,14,
```

```
POKE PEEK (138)+256*PEEK
(139)+2,0:SAVE''D:FILE-
NAME'':NEW
```

It does not matter if the line is at any time executed by the main program; the code therefore remains unaffected in any way. It is imperative, however, that the line be the chronologically last line of code. When you are ready to protect a program (that is, do not intend to alter it any further), type this line with a higher line number than any other in the program, choose a filename, than GOTO the line. Listing 2 is a working example.

That is all there is to the technique: "RUN only" files can be simply generated to disk or tape. Attempts to do anything other than RUN will result in a nasty case of system lock-up. And yes, even autorun files can be protected in this manner.

Scuttlebytes

Well, we have finally managed to confirm the existence of the *Atari 600,* and have heard that at least two Atari plants are currently tooling up to produce them. The 600, as its model number implies, will fill the gap between the Atari 400 and 800., It was rumored that the machine would be unveiled at the Winter CES in Las Vegas. It will sport 48K standard, and a full-stroke keyboard. Owners of 400s and 800s need harbor no fears of obsolescence: the 600 will be completely compatible with its predecessors.

Many Atari types are awaiting with curiosity the final verdict on the Commodore 64, which features graphics, sound, and gaming capilities very much akin to those of Atari computers. Atari has, in the meantime, added to its busy legal docket a suit against Commodore, concerning the design implementation of Commodore joysticks, which are for use with the VIC-20 and all latest generation machines. It seems the sticks are not only Atari-compatible, but nearly identical in many respects.

Atari, which patented its stick when the VCS was first introduced in 1977 and improved the design several times since, claims patent infringement. The Atari joystick connector has set an informal design standard in the industry. The Colecovision videogame uses an Atari-compatible format, and it was rumored that the new microcomputer, to be released by Apple this year will also make use of Atari-compatible digital sticks. But compatibility and patent infringement are two separate concepts.

Joysticks

While we're on the topic of joysticks, let me tell you about two hot sticks

```
13,2,13,2,19,17,19,25,11,F,25,9,F,17,1
,F,2,1
866 DATA P,16,9,F,16,11,F,14,13,F,2,13

870 REM "H"----------------------------
-
872 DATA R,35,47,5,1,172
874 DATA P,2,1,2,7,19,7,16,10,19,13,2,
13,2,19,25,19,25,13,F,22,10,25,7,25,1,
F,2,1
876 DATA P,16,10,F,19,7,P,2,13,F,19,13

880 REM "X"----------------------------

882 DATA R,35,69,5,2,191
884 DATA P,2,1,2,7,9,10,2,13,2,19,8,19
,13,15,19,19,25,19,25,13,F,18,10,25,7,
25,1,F,18,1,F,13,5,F,8,1,F,2,1
886 DATA P,10,10,F,2,13
890 REM "Y"----------------------------
-
892 DATA R,35,91,5,3,205
894 DATA P,2,1,2,7,9,10,2,13,2,19,8,19
,13,15,16,13,25,13,25,7,F,16,7,F,13,5,
F,8,1,F,2,1
896 DATA P,10,10,F,2,13
900 REM "Z"----------------------------
-
902 DATA R,35,113,5,0,255
904 DATA P,2,1,2,19,8,19,19,10,19,19,2
5,19,25,1,F,19,1,F,8,10,8,1,F,2,1
1000 DATA END
1010 RESTORE 1040
1020 FOR I=1 TO 32:READ C:C$(I)=CHR$(C
):NEXT I
1030 RETURN
1040 DATA 104,104,104,72,162,57,160,0,
173,0,210,101,20,141,22,208,141,10,212
,136,208,242,202,208,237,104
1050 DATA 56,233,1,208,228,96
```

Listing 2.

```
10 ? "THIS PROGRAM IS UNLISTABLE"
20 ? "EVEN THOUGH THE RESET AND"
30 ? "BREAK KEYS REMAIN ENABLED."
40 ?
50 ? "TRY IT!"
60 FOR X=1 TO 1000:NEXT X
70 GRAPHICS 0:GOTO 10
80 REM REMEMBER TO "GOTO 10000"
90 REM TO SAVE THE UNLISTABLE FILE!
10000 POKE PEEK(138)+256*PEEK(139)+2,0
:SAVE "D:OUTPOST":NEW
```

we've been playtesting. The first is the Pointmaster joystick. This stick has an extra long handle with built-in grip and handle-mounted trigger button, making it perfect for "flyer" games like *Star Raiders* and *Protector II.* The stick is very much like the one in the stand-up arcade version of Zaxxon, and once you play a few games of *Raiders* with it, you won't want to use anything else. Conversely, the stick is cumbersome in maze games like *Jawbreaker* or MBAFAS ("move back and forth and shoot") games like *Threshold.* Still, at $17.95, it offers a real boost to your "flyer" game collection.

For more information, contact Discwasher, 1407 North Providence Road, Columbia, MO, 65201. (314) 449-0941.

The other sticks we looked at, called Game Mate 2, are pretty nearly regulation Atari sticks, with one big difference: they are wireless, and work by remote control. My main fear was that there would be a time lag between the movement of my hand and what I saw on the screen. I experienced no such sensation—the sticks seemed as fast as any I had ever tried. My only reservation is their size. They are quite bulky, and take a while to get used to.

With the VCS, the console power supply plugs into the Game Mate receiver unit, and then into the VCS console. For the 400 and 800 computers, however, an additional 9-volt power supply is a necessary purchase. Each stick also takes a 9-volt transistor battery. The units operate at distances of up to 20′.

Complete with receiver and two sticks, Game Mate 2 lists for $99.95, but I have already seen this price substantially discounted. If the luxury of wireless sticks is appealing to you, this product will assuredly not disappoint.

For more information, contact Cynex Manufacturing Corporation, 28 Sager Pl., Hillside, NJ, 07205. (201) 399-3334.

Games

Smoothly we seque from sticks to the games played with them. We have confirmed 400/800 versions of *Galaxian* and *Defender* in ROM form from Atari. Both games are spin-offs from the new 5200 model videogame. The 5200 may yet prove to be a boon to owners of Atari computer systems, if it spurs game development common to all machines. *Galaxian* has already been demonstrated, and is a solid implementation. One can only hope that *Defender* will be up to snuff.

Datasoft also has an ambitious project on its drawing boards right now: *Zaxxon* for the Atari. We can't wait.

I must admit it: when I first heard that Big Five Software was releasing a game for Atari, I sort of chuckled. Somehow I assumed that because *Miner 2049*'er was from one of the best TRS-80 game houses, it would probably run in graphics 5. Did I make a mistake. *Miner 2049'er*, in ROM cartridge format, is bound to be one of the runaway hits of the year. With superlative graphics, humor, and 10 completely different screens to master, the game leaves Colecovision's *Donkey Kong* pale by comparison. □

The Challenge is Met

In my "Super Text Mode" article, I posed a challenge to all Atari hackers for vast improvements upon my program. The first response I received was from Mike Portuesi, a sixteen-year-old Atari devotee from Mount Clemens, Michigan.

Mike succeeded in the task with satisfyingly little code. His version of the program accepts a user message as a string, modifies itself to include only the letters needed for that specific message, then deletes all extraneous lines. The user needs only to reposition those letters on the screen, which is a very simple process. Tinkering with sound values and plotting speed will result in a polished title card. Creating multiple cards is made dramatically less time-consuming.

The additions appear below (these lines must be added to the program appearing on pages 142-145).

For a bit of background and a walk through the modifications, I now turn things over to Mr. Portuesi:

Operation of the program is simple. Simply RUN it, and the program will ask you to input your message. Use no blanks, please. There is a 21 character limit, because I figure that 21 characters are the most you can fit on the screen at once (3 x 7), but if you're using lots of skinny letters (like "I"), simply change the DIM statement at line 5. When it finishes running, you are left with a customized program.

The main challenge I faced in the modification was renumbering the data statements to fill the gap between line numbers 230 and 650. I couldn't live with repositioning the cursor and printing new line numbers. I would have gone insane trying to come up with a routine to account for missing lines and different line lengths. My program uses a different method, as follows:

1. Get message
2. For each character of string:
   a. list to screen all associated data lines
   b. Use forced-read mode to input lines into A$, B$, C$, and D$
   c. Modify strings to set new line numbers
   d. Print strings on screen
   e. Force-read them into the existent program
3. Delete lines 650 to 904
4. Delete lines 5 and 859, then RESTORE data pointer to line 232
5. Delete modification routine itself.

The forced-read mode is used not only to modify the program, but also to enter program lines into a string.

Here is a line by line explanation of the added lines:

5 — DIM strings, GOTO 2000
2000-2010 — Get message from user
2020 — Loop for each character in the string
2040-2050 — List all lines relating to a specific letter
2060 — Enter these lines into A$, B$, C$, and D$, with forced read mode (using INPUT, not STOPping the program)
2070-2100 — Modify A$, B$, C$, D$, so as to give them new line numbers
2110-2120 — Put these lines back out on the screen
2150-2190 — Delete lines 650 through 904, twenty lines at a time
2220 — Delete lines 5 and 859, change line 120 to RESTORE pointer
2240 — Delete first half of modifcation routine
2250-2260 — Delete rest of modification routine, stop program for user
3000-3030 — Modification subroutine

A really big problem I had in development of the program is the infamous keyboard lock-up that occurs with repeated and heavy editing. This, coupled with the fact that I have only a cassette recorder, led to heartaches and frustration. I wish **somebody** would do something about that. For all the user-friendliness of Atari Basic, that bug almost makes me want to take the Basic cartridge, squirt it down with lighter fluid, and take a match to it.

One quick word of warning: remember to SAVE Mike's additions to the program before ever RUNning the modified program! As soon as it runs, it deletes the powerful parts of itself. Skip any testing until you put a file on disk. Otherwise, you too may look for the lighter fluid.

Listing 3.

```
5 DIM STRING$(21),A$(120),B$(120),C$(120),D$(120):GOTO 2000
2000 ? CHR$(125):? "PLEASE ENTER YOUR STRING":? "(LESS THAN 21 CHARACTERS, PLEAS
E!)"
2010 INPUT STRING$:LINENO=232
2020 FOR I=1 TO LEN(STRING$)
2030 ? CHR$(125):POSITION 2,2
2040 FOR J=ASC(STRING$(I,I))*10+2 TO ASC(STRING$(I,I))*10+8 STEP 2
2050 LIST J:PRINT "":NEXT J:POSITION 2,3
2060 POKE 842,13:INPUT A$,B$,C$,D$:POKE 842,12
2070 A$(1,3)=STR$(LINENO):LINENO=LINENO+2
2080 B$(1,3)=STR$(LINENO):LINENO=LINENO+2
2090 C$(1,3)=STR$(LINENO):LINENO=LINENO+2
2100 D$(1,3)=STR$(LINENO):LINENO=LINENO+2
2110 ? CHR$(125):POSITION 2,2
2120 ? A$:? B$:? C$:? D$
2130 GOSUB 3000
2140 NEXT I
2150 PNTR=1:? CHR$(125):POSITION 2,2
2160 FOR I=650 TO 904 STEP 2
2170 ? I
2180 PNTR=PNTR+1:IF PNTR=20 THEN PNTR=1:GOSUB 3000:? CHR$(125):POSITION 2,2
2190 NEXT I
2200 GOSUB 3000
2210 ? CHR$(125):POSITION 2,2
2220 ? 5:? "120 RES. 232":? 859:GOSUB 3000
2230 ? CHR$(125):POSITION 2,2
2240 FOR I=2000 TO 2180 STEP 10:? I:NEXT I:GOSUB 3000
2250 ? CHR$(125):POSITION 2,2:FOR I=2190 TO 2260 STEP 10:? I:NEXT I
2260 FOR I=3000 TO 3030 STEP 10:? I:NEXT I:? "POKE 842,12:?CHR$(125)":GOTO 3000
3000 ? "CONT":POSITION 0,0
3010 POKE 842,13:STOP
3020 POKE 842,12
3030 RETURN
```

Converting Applesoft Basic to Atari Basic

Neater Numerical Tables

Paul N. Havey

In *Neaten Up Those Messy Numerical Tables* by Donald J. Taylor (*Creative Computing,* December, 1981), the author designed a short program, written in Applesoft, which enables the user to print numbers in a neat exponential form. While the program may be an answer to the dreams of many an Apple owner, it is of little use to still-dreaming Atari owners.

Conversion from Applesoft to Commodore or TRS-80 Basic is more direct than conversion to Atari Basic. This is because Atari Basic does not support string arrays like the others. However, with the unlimited string length capability of Atari Basic, program conversions are possible for Atari owners and, in some cases, simplified.

Included here is a translation from Applesoft Basic to Atari Basic of Taylor's exponential program. Also included is a listing of the original program for comparison. Both programs convert a number to a string which is then disassembled. The number is then reassembled by concatenation into a thousand string mantissa and exponential for the scientific notation format. Zeros and signs are inserted where needed. See Table 1 for a sample of printed output.

The variable names and time requirements for the Atari Basic version are only slightly different from the original Applesoft Basic version. Table 2 contains a list of variable names for the Atari Basic version and Table 3 contains the time and memory requirements for both versions.

The handling of string variables is one of the fundamental differences between Applesoft and Atari Basics. In Microsoft Basic (and in Applesoft) extraction or splitting a string into pieces is done by the functions MID$, RIGHT$, and LEFT$. In Atari Basic, strings are split by using a subscript or set of subscripts. For example, A$(6, 12) means that the substring starts at the sixth character and ends with the twelfth. If one number is within the subscript then the subscript

Paul N. Havey, P.O. Box 5148, Santa Monica, CA 90405.

Program 1.

```
10 REM SCIENTIFIC NOTATION FORMAT
20 REM CONSTANT SIGNIFICANT DIGITS
30 REM TRUNCATOR/ROUNDER
40 REM REVISED FOR ATARI BASIC
45 REM BY PAUL HAVEY  01 DEC 82
50 REM INPUT: TWO VARIABLES
60 REM 1. X=NUMBER TO FORMAT
70 REM 2. D=SIGNIFICANT DIGITS
75 REM    D-1 FOR ROUNDING
80 REM OUTPUT: ONE VARIABLE
90 REM 1. X$=FORMATTED NUMBER
100 DIM X$(20),BUF$(20),E$(3),BUFI$(10),S$(1)
110 S$=" ":IF SGN(X)=-1 THEN S$=" "
120 BUF=ABS(X):BUF$=STR$(BUF):L=LEN(BUF$)
130 IF L>4 THEN IF BUF$(L-3,L-3)="E" THEN E$=BUF$(L-2):
    BUF$(L-3)="0000":GOTO 230
140 IF BUF=0 THEN X$=" 0":GOTO 300
150 IF BUF<1 THEN 210
160 BUFI$=STR$(INT(BUF))
170 E$="+0":E$(3,3)=STR$(LEN(BUFI$)-1)
180 IF BUF<10 THEN 230
190 BUF$(2,2)=".":BUF$(3,LEN(BUFI$)+1)=BUFI$(2)
200 GOTO 230
210 IF BUF$(3,3)="0" THEN E$="-02":BUF=BUF*100:
    BUF$=STR$(BUF):GOTO 230
220 E$="-01":BUF=BUF*10:BUF$=STR$(BUF)
230 BUF$(LEN(BUF$)+1)="00000000"
240 BUF$=BUF$(1,D+1):BUF$(2,2)="."
290 X$=S$:X$(2)=BUF$:X$(LEN(X$)+1)="E":X$(LEN(X$)+1)=E$
300 RETURN
```

Program 2.

```
250 REM ROUNDING ROUTINE
260 BUF$(2)=BUF$(3):BUF=(VAL(BUF$)+5):BUF$=STR$(BUF):
    IF LEN(BUF$)=D THEN 280
270 L=ABS(VAL(E$)+1):E$(2)=STR$(L):IF L<10
    THEN E$(3)=E$(2):E$(2,2)="0"
280 X$=BUF$(2):BUF$(2,2)=".":BUF$(3)=X$(1,D-2)
```

Table 1. Sample Data Columns.

Column A	*Column B*
-4E-04	-4.00E-04
-1.10679718E-03	-1.11E-03
-0.02	-2.00E-02
-0.158113883	-1.58E-01
0	0
1.58113883	1.58E+00
10	1.00E+01
47.43416949	4.74E+01
200	2.00E+02
790.569415	7.91E+02
40000	4.00E+04
1739252.71	1.74E+06
2.68793601E+09	2.69E+09
9E+09	9.00E+09

A = As the Apple and Atari print.

B = As converted by this program (rounded off to three significant digits).

begins with that character and ends with the last character in the string.

Table 4 contains a list of Atari to Microsoft translations. The LEN(A$) function is the same in both types of Basic.

Atari Basic uses the LEN(A$) functions to concatinate two substrings. In Microsoft Basic, concatination is done with a plus sign. Table 4 shows an example of each.

Microsoft Basic uses subscripts as an indication of a string array. Atari Basic, while not supporting string arrays, can simulate string arrays by using subscripts. In Atari Basic, simulated arrays that have all the same length are the most useful.

Program 3.

```
100 REM SCIENTIFIC NOTATION FORMATTER
102 REM CONSTANT SIGNIFICANT FIGURES
104 REM TRUNCATOR/ROUNDER
106 REM DONALD J. TAYLOR
108 REM JUNE 1, 1982
110 REM INPUTS REQUIRED=X AND D
112 REM X=INPUT #
114 REM D=# DIGITS FOR TRUNCATION
116 REM D-1=# DIGITS FOR ROUNDING
118 REM X$=OUTPUT
120 REM
122 S$=" ":IF SGN(X)=-1 THEN S$="-"
124 X=ABS(X):X$=STR$(X):P=LEN(X$)
126 IF P>4 THEN IF MID$(X$,P-3,1)="E" THEN P$=MID$(X$,P-2)
    :X$=LEFT$(X$,P-4):X$=LEFT(X$,1)+"."+MID$(X$,3):GOTO 140
128 IF X=0 THEN X$=" 0":P$=" ":GOTO 164
130 IF X<1 THEN GOTO 136
132 XI$=STR$(INT(X))
134 X$=LEFT$(XI$,1)+"."+MID$(XI$,2)+MID$(X$,LEN(XI$)+2)
    :P$="+0"+STR$(LEN(XI$)-1):GOTO 140
136 IF MID$(X$,2,1)="0" THEN P$="-02":X$=MID$(X$,3,1)+"."+
    MID$(X$,4):GOTO 140
138 P$="-01":X$=MID$(X$,2,1)+"."+MID$(X$,3)
140 X$=X$+"00000000"
142 X$=LEFT$(X$,D+1)
162 X$=S$+X$+"E"+P$
164 RETURN
```

Program 4.

```
144 REM ROUNDING ROUTINE : LINES 144-160
146 X$=LEFT$(X$,1)+MID$(X$,3)
148 X=VAL(X$)+5:X$=STR$(X)
150 IF LEN(X$)=D THEN GOTO 160
152 P=VAL(RIGHT$(P$,3))+1
154 IF SGN(P)=1 THEN GOTO 158
156 P$="-"+RIGHT$(("0"+STR$(ABS(P))),2):GOTO 160
158 P$="+"+RIGHT$(("0"+STR$(ABS(P))),2)
160 X$=LEFT$(X$,1)+"."+MID$(X$,2,D-2)
```

Table 2. Atari version of the variable list.

Name	*Description*
S$	Sign of original number (+ or –)
E$	Exponent part of number (E)
BUF$	Mantisa part of number
BUFI$	Integer mantisa part of number
BUF	Original unformatted, unsigned number
L	Length of mantisa part of number
X$	Final version of number for printing
X	Original unformatted, signed number

The original number "X" is disassembled into the strings "$$", "E$", and "BUF$". After processing, the number is reassembled as the string "X$".

Table 3. Atari Time and Memory Requirements.

Duration with rounding in milliseconds:	170
Duration without rounding in milliseconds:	125
Memory used in bytes:	1400

Table 4. String Operation Comparisons.

Item #	*MICROSOFT*	*ATARI*
1	MID$(A$, X,Y)	A$(X,Y)
2	LEFT$(A$,X)	A$(1,X)
3	RIGHT$(A$,X)	A$(LEN(A$)-X)
4	A$=A$+B$	A$(LEN(A$)+1)=B$
5	C$=A$+B$	C$=A$:C$(LEN(C$)+1)=B$
6	A$(1)="AAA"	A$(1,3)="AAA"
7	A$(2)="BBB"	A$(4,6)="BBB"
8	A$(3)="CCC"	A$(7,9)="CCC"

Table 4 contains the most common string operations for both Basics. Items 1, 2, and 3 perform substring extraction. Items 4 and 5 concatenate two strings. Items 6 to 8 give examples of string array notation.

Interfacing Your Atari

Marshall S. Dubin

Looking for some "off the beaten path" type of excitement? Tired of blasting aliens, running through mazes, or balancing your checkbook? Are you the adventuresome type? Well, this could be the project for you! With a few parts and a little time in the workshop, you can have your Atari lighting lights, dialing phones, reading and regulating thermostats, and generally communicating with the outside world.

Through the front controller ports of the Atari computer, there are available for your use 16 programmable input/output pins, 8 analog to digital inputs, and 4 input only pins. These controller ports can be used with interface circuitry to monitor "real world" devices such as thermostats or light sensors, or to activate relays, motors, and lights.

We'll discuss various ways of using the front controller parts to communicate with the outside world. For the braver of you, we will be building an I/O interface, so that you may sense signals, and/or turn on small relays. This will plug into the front port of your computer and allow you to connect various real world devices. (VIC owners should note that the joystick ports on your machine are identical to the Atari ports. With the exception of any software drivers, the electrical connections should be the same.)

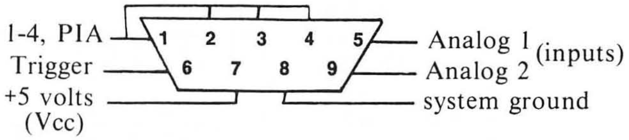
Please note that this kind of interfacing may void your warranty. If you are not sure, then check with your dealer or factory representative. Also note, that accidents DO happen. It is possible (although unlikely if you are careful) to do some drastic damage to your computer. If you are not sure how to do something then DON'T DO IT. Also keep in mind that although the power required for digital work is usually between 5 and 12 volts, a relatively safe level, the primaries of these power supplies are usually 110 volts. Follow these few common sense rules:

1. Be careful. Always keep high voltage well away from your work area.
2. Be neat. Lots of wires scattered around tend to short something out.
3. Never do any wiring or soldering with the power turned on.
4. Use a low heat (25 watt) soldering iron. Do not use a soldering gun.

Now let's do some interfacing!

Marshall S. Dubin, 2639 Hempstead, Auburn Heights, Michigan 48057.

Figure 1. Front panel pin diagram.



The Basics

As you can see from the pin diagram in Figure 1, each joystick port has several potential input sources available. For example, two of the pins are intended for use with the paddle controllers. These are called the Analog pins. They take an analog source such as a variable resistance and convert it into a digital signal. This is in essence how the paddles function. They provide a resistance via a potentiometer within the paddle unit, between the analog input pins and +5 volts DC. The computer interprets the variable voltage as a digital number between 0 and 228. This is called "on board" analog to digital conversion. Units performing a similar function may be purchased at a hefty price, but Atari owners have the use of 8 of these units built right in!

For now, let's concentrate on pins 1-4 on the joystick ports. These are the pins of the *Peripheral Interface Adapter* chip, more commonly referred to as the PIA. Basically the PIA provides a means of connecting your computer to peripherals. The PIA chip can be programmed for either input or output. There are two PIA ports of eight bits each available for your use. Joystick ports 1 and 2 compose PIA port A, while joystick ports 3 and 4 compose PIA port B. Each port is one byte (8 bits) and may be used together or individually to provide input and output functions. Some of these functions may be used to drive a printer or other accessory, or even a series of power relays which can control alarms, lights, appliances, motors or other device.

The snag involved in controlling larger interface devices is basically a problem of taking a small amount of power and amplifying it. The ports on your computer are not made to power anything more than another chip. The manual recommends a maximum of 1 TTL load (about 1 chip) at 50 ma *tops.* To be of any real use, we must be able to power at least 12 to 24 volts. This higher voltage can drive a wide variety of relays and interfaces.

There are several ways to accomplish this task. The most common arrangement is the transistor driver. In this arrangement the computer provides a very small voltage which turns on the transistors which in turn switch the load. A second way is through the use of opto-isolators. The computer provides 5 volts which switches the LED (light emitting diode) of the isolator. When the diode is lit, this triggers a photo sensitive transistor which is connected to a larger load or a relay.

A third way, and the one we shall use, is to employ an integrated circuit interface chip. The chip we will be using is the SN7407 made by Texas Instruments. The 7407 allows a switching of up to 30 volts from the 5 volt TTL level of the Atari, with enough current to handle a small relay. Using this one chip, we can drive up to six relays from the Atari front ports.

The SN7407, as shown in Figure 2, is an open collector device. To use it properly you must connect a 2.2K ohm resistor from each output to +5 volts. This is called a "pull up" resistor. When an output of the 7407 is "on" it is actually open — so the resistor supplies power to the device you are driving. You can drive up to 30 volts at the outputs (but you may have to tamper with the value of the resistor somewhat). When an output is "off", it is shorted to ground, and your device sees 0 volts (ground actually). The resistor limits this current to a fairly low value so you don't blow the power supply or worse, the chip! Now this is the sequence of events:

Atari: HIGH (logic 1)

7407: OFF —device is OFF.

Atari: LOW (logic 0)
7407: ON —device is ON.

Since the resistor can't supply much current, the resistor/7407 combination is seen as the "ground side" of the circuit. That is, to drive a relay, we connect power to one side of the relay, and the other side to the output of the 7407. Then when we turn the relay on, Current will flow through the relay, and then through the 7407 to ground.

You can easily drive LED's this way too (such as for test lights), as well as a variety of small relays or solid state switches. Just make sure you SINK the current — that is, one end of your driven device goes to +5 (through a resistor!) and the other end to the 7407. Sending a "0" (logic level low) to the PIA turns the device ON, and a "1" (logic level high) turns it OFF. If you want to do it the other way around, use the inverting 7406 chip, which will turn your device ON with a high logic level and OFF with a low level. Recognize that the default state of the PIA when the computer is powered up is all bits high. If you are using an inverting 7406, your devices would come alive when you powered on the Atari. This is why I prefer to use the 7407, since I can power up and then have my software drive the devices by writing a 0 to the bit I want to power a device from.

Speaking of bits, a few words are in order about the structure of the ports before you run off to warm up your soldering irons. The PIA as I mentioned earlier consists of two ports, port A and port B (or PORTA and PORTB for all you memonic freaks). These are controlled through the use of the control registers for each port, PACTL and PBCTL. You may have heard of the PACTL because that's the one you POKE to turn on the cassette player. The addresses are as follows:

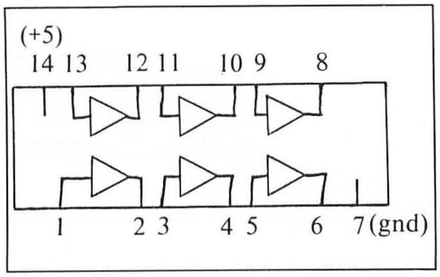
PORTA 54016/$D300 —
port A address

PORTB 54017/$D301 —
port B address

PACTL 54018/$D302 —
port A control

PBCLT 54019/$D303 —
port B control

Figure 2. SN 7407 diagram.



On power up, the ports are initialized to $FFFF or all bits high. To use a port for input, just pull the bit of your choice low by connecting it to ground. To use the port for output, it first must be formatted for output. The procedure is not complex:

1. POKE the control register (PACTL or PBCTL) with 56/$38 hex.
2. Now poke the port (PORTA or PORTB) with 255/$FF hex. This specifies the port will be used for output.
3. Poke PACTL or PBCTL with 60/$3C hex.
4. Now just poke the port (PORTA or PORTB) with your data.

Essentially you have a total of 16 bits to play with. Just remember that two joystick ports make up one PIA port. Stick 0 and 1 are the A side and stick 2 and 3 are side B. Each joystick port is 4 bits or 1 nibble. Each side of the PIA is 8 bits or 1 byte. When programming for output, you must remember that a specific BIT is driving a device. Therefore one joystick port can drive 4 devices (1 for each bit). An entire PIA side will handle 8 devices and if you use both A and B sides you can trigger 16 individual devices at once or in any combination. You must POKE into that port a decimal number whose BINARY representation will switch on a certain bit or series of bits. For example, if I POKE a 255 into port A, all bits would be on. If I POKE a 12 into port A, bits 3 and 4 only would be on, since the binary of 12 is 1100 The individual joystick ports may be read using the shadow registers as follows:

Jack 1 (STICK 0) 632/$278 hex

Jack 2 (STICK 1) 633/$279 hex

Jack 3 (STICK 2) 634/$27A hex

Jack 4 (STICK 3) 635/$27B hex

Each port will return a number between 0 and 15. You also can use the BASIC keywords STICK to access these ports eg. X=STICK(0), etc.

The Hardware Part

Generally, all of the circuits we will describe can be breadboarded in any way convenient for you. For those of you just starting out, and who want to do some experimenting, I recommend the following workbench supplies:

— a solderless breadboard or wire-wrapping set-up for prototyping your circuitry. (Such as those made by Vector, Tandy, etc.)

— a variety of IC and transistor sockets

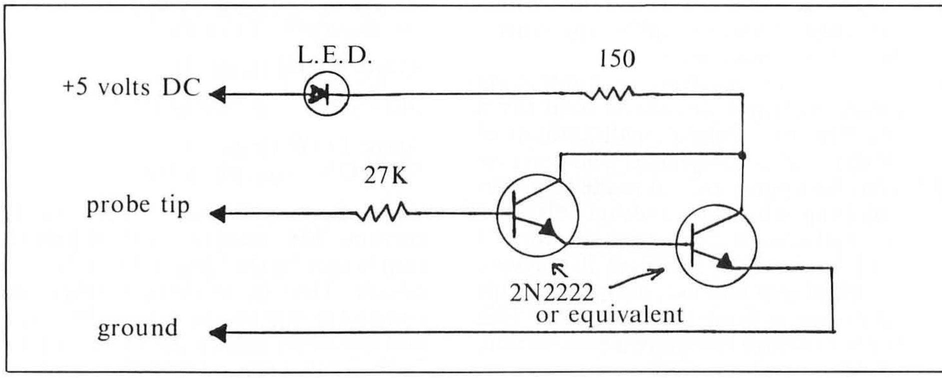
— an anti-static mat or spray

In addition, for this project, you will need at least one DE9S connector to match the front joystick port, and some multi-conductor wire.

Now let's get started by building another useful tool for you to use: a logic probe. Figure three shows the construction diagram of a two-transistor logic probe. You can "steal" the necessary 5 volts from the computer on board power supply. When

Pin	*Function*
1	gate 1 in
2	gate 1 out
3	gate 2 in
4	gate 2 out
5	gate 3 in
6	gate 3 out
7	ground
8	gate 6 out
9	gate 6 in
10	gate 5 out
11	gate 5 in
12	gate 4 out
12	gate 4 in
14	+5 volts DC (VCC)

Figure 3. Logic Probe Schematic.

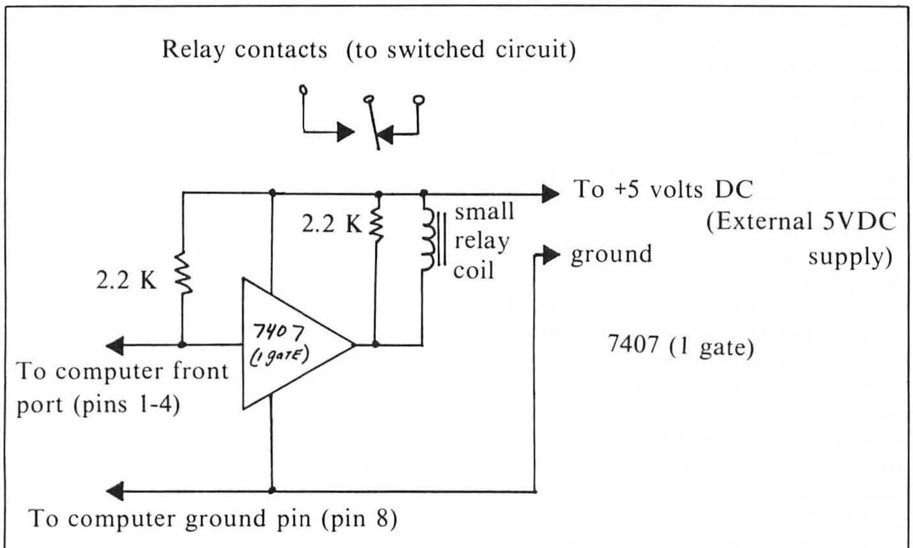


```
10 REM * PROGRAM TO FORMAT PIA PORTS
20 REM
30 GRAPHICS 0:POSITION 10,2
40 DIM IO$(10),DATA$(3)
50 PRINT "PIA PORT DEMO"
60 REM
70 REM PORT ADDRESS
80 REM
90 PORTA=54016:PORTB=54017
100 REM
110 REM * ROUTINE TO CONFIGURE PORT
120 REM
130 TRAP 130:PRINT :PRINT "Configure which port (1-4) ";
140 INPUT PORT:IF PORT<1 OR PORT>4 THEN 130
150 REM
160 REM SELECT PORT CONTROL REGISTER
170 REM ADDRESS (PACTL,PBCTL)
180 REM
190 IF PORT<3 THEN PCTL=54018:PORT=PORTA
200 IF PORT>2 THEN PCTL=54019:PORT=PORTB
210 PRINT :PRINT
220 REM
230 REM SELECT INPUT OR OUTPUT
240 REM
250 PRINT "Input or Output ";
260 TRAP 250:INPUT IO$
270 IF IO$(1,1)="I" THEN F=0:GOTO 340
280 IF IO$(1,1)="O" THEN F=255:GOTO 340
290 GOTO 250
300 PRINT
310 REM
320 REM CONFIGURE THE PORT
330 REM
340 POKE PCTL,56
350 POKE PORT,F
360 POKE PCTL,60
370 PRINT :PRINT
380 REM
390 REM ENTER YOUR DATA
400 REM
410 IF  IO$(1,1)="I"  THEN  PRINT  "PORT  IS  FORMATTED
    FOR INPUT":PRINT :GOTO 130
420 PRINT "NOW ENTER YOUR DATA"
430 PRINT "(ENTER A RETURN TO DO ANOTHER PORT)"
440 INPUT DATA$:IF DATA$="" THEN PRINT CHR$(125):GOTO 130
450 TRAP 530
460 REM
470 REM POKE DATA TO PORT/VERIFY IT
480 REM
490 POKE PORT,VAL(DATA$)
500 PRINT "VERIFY ";PEEK(PORT)
510 GOTO 440
520 END
530 TRAP 40000:PRINT "INPUT ERROR, RE-ENTER ";:GOTO 440
```

the LED is on this indicates a logic 1 or high condition. No LED indicates a 0 or low. (Actually that is not exactly true. This logic probe cannot detect the actual "0" state. There are more sophisticated probes able to differentiate high, low, and high impedance logic states — but what do you want for less than a buck?)

One potential way to mount the probe is to build the unit on a small (1″ by 2″) perfboard, and then slide the completed assembly into a large cigar tube. A small probe tip could then be soldered to the front of the cigar tube, and wires for the required 5 volts and ground could come out the back end. These would terminate in small alligator clips.

Figure 4. 7407 Interface (1 gate).



To use additional gates, connect them as shown. You only need to connect ground (pin 7-7407) and +5 (pin 14-7407) once to power all gates.

A Useful Interface Board

Figure four is a schematic diagram of an output interface board which is connected to the front ports of the computer.

The heart of the circuit is the 7407 chip. As you can see, the input channels of the 7407 are connected to the front port by way of the 2.2K pull-up resistor. Even though the port initializes to FFFF (or all ones), this maintains a high state until we do otherwise.

The output of the 7407 provides the ground side of a relay circuit. The relay coil is connected to the 5 volt supply (NOT the one on the computer port!). When the 7407 is activated by having one of the front port pins pulled low, it provides the relay coil with a path to ground, and the coil energizes. The relay controlled by the coil can be used to switch on just about anything, including other relays to drive larger loads.

During construction be careful that all the pins of the 7407 and the components are wired correctly. Do not forget the pull-up resistors or the small capacitors. These help prevent power supply interference. If you wish, you can substitute small LED's or 5 volt pilot lamps for the relays. This will allow you to see the ports in action.

Be sure to use an external power supply or 5 volt source. The ground of your source should be connected to the ground pin on the computer. You will not need the 5 volt pin. Incidently, you can use the external supply to power your logic probe, and still read the computer logic signals at the ports.

Now For a Little Software

The program listing will provide you with a demonstration on how the ports are programmed. The program first allows you to select a port, and program it for either input or output. Then you can write data to the port and the computer will peek the port and verify the data you wrote. You can also do this by using the logic probe. You will get a logic 1 for every active bit in the port.

Atari Strings and Text Handling

Apples, Oranges, TRS-80s and the Atari

The Atari, unlike the Pet, TRS-80, Apple, and Heathkit computers, does not have a Basic by Microsoft. This is a mixed blessing, or mixed curse, as you choose to look at it. The graphics and music handling abilities of Atari Basic are a true joy, while the string handling is difficult. Since most programs published in computer magazines like this one are not in Atari Basic, an understanding of the differences is helpful if you wish to convert the programs for your own use. Here are some of the differences between the Atari and the TRS-80, the most common Microsoft Basic computer.

String Handling

In Atari Basic DIM A$(50) means reserve 50 bytes of memory for a single variable named A$. You cannot store a single letter in a string variable unless you dimension it first. One advantage of this is that you can control the length of a string just by the DIM statement, something you cannot do in Microsoft Basic. For example, if you put:

```
10 DIM ANSWER$(1)
```

Then the computer will store only the first letter in the string even if it receives a whole sentence as a reply. This makes it easy to test an answer:

```
20 PRINT“ANSWER”;:INPUT
ANSWER$:IF ANSWER$=“N”
THEN50
```

In the TRS-80, memory for all string variables is reserved by a single CLEAR statement, with a default value of 50 bytes reserved automatically even without a CLEAR statement. In the TRS-80, DIM A$(50) means create an array of 51 string variables from A$(0) through A$(50).

In the TRS-80, the maximum length of a string ranges from 241 to 256 bytes, depending on circumstances. The Atari is limited only by memory available. This means that the Atari can make up for the lack of string arrays through a process of storing substrings in a very long string. One advantage of the Atari is that string sorting is potentially faster, as the TRS-80 has to pause and reorganize its string space.

Related to the string length is the restriction of a TRS-80 program line to 241 to 255 characters, while the Atari observes a different approach and limits you to 120 characters. Since some TRS-80 programmers like to put a whole subroutine in a single line, you would have to do a bit of reshuffling to translate their programs to an Atari. The lack of an ELSE command further restricts this approachin the Atari.

Sound

Neither the Atari nor the TRS-80 have a built-in speaker, as does the Apple II. The Atari sends sound effects through a television set speaker. If you are using a monitor that does not have a speaker, you do not have sound. Common practice with the TRS-80 is to connect an amplifier to the cassette output port.

The real difference in sound is that the Atari has a built-in sound capability allowing four completely separate voices at the same time with over 20,000 sound options, including a wide range of musical notes for each voice, while TRS-80 Basic can only alternate voltages at the cassette output port with OUT statements or machine language subroutines. Harmony is very difficult with the TRS-80, but easy in the Atari.

Graphics

It is not really fair to compare TRS-80 graphics to the Atari, as the TRS-80 is strictly medium resolution black and white while the Atari has high resolution color. To fairly represent Microsoft Basic, the Apple should be included in the discussion. One advantage the TRS-80 does enjoy is easy mixing of text and graphics on the screen, which is more difficult with the Apple and the Atari. Also, the TRS-80 has a built-in video memory that does not require user memory, while the Apple and Atari require user memory and, in high resolution, lots of it.

The Atari has 16 different graphics modes, and some of the graphics in the Atari ROM cartridges, including the motion through space in Star Raiders and the ability of the basketball players to overlay each other in Basketball, promise more graphics power than any other popular home computer. Right now, a side-by-side comparison of Apple and Atari graphics seems a standoff because the Atari graphics are not yet documented and explained, but if this kind of graphics ability becomes accessible to the end user, the Atari will be the obvious choice.

A common problem in high resolution graphics is that it requires a lot of memory to store a detailed image. The normal sacrifice limits the number of colors available in hi-res so you need less memory to store color information. The Atari limits you to two colors in high resolution, while the Apple gives you four. However, the Atari allows you to choose your color and tint and even allows you to change the color of an image on the screen instantly by changing a color register that tells the computer what color to make the image. The Apple cannot match this ability.

My personal favorite among the graphics commands of the Atari is the DRAWTO statement, which draws a line from the last plotted point on the screen to any other point. More or less the same ability is present in the HLIN and VLIN commands in the Apple, though not as easily, nor as fast. In the TRS-80, it is necessary to write a subroutine to plot each point individually.

Text Handling

Text handling in the Atari is not as convenient as the Microsoft Basics. The TRS-80 is particularly good at text formatting and printing. Microsoft Basic allows you to include text in an INPUT statement, like this:

```
10 INPUT“What is your answer”;A$
```

Atari Basic requires a separate print statement:

```
10 PRINT“What is your answer”;:
INPUT A$
```

The TRS-80 allows you to print directly at any point on the screen with PRINT @:

```
10 PRINT @ 572,“X MARKS THE
SPOT”
```

The Atari requires you to position the cursor first, then print your message:

```
10 POSITION 8, 12:PRINT
“X MARKS THE SPOT”
```

Still another difference in text handling is the power of Microsoft Basic's PRINT USING command, allowing you to specify automatic print formatting with a fixed number of decimal points, floating dollar signs, fixed spacing, and other conveniences. These things have to be done by manipulating a string in Atari Basic.

I have begun to experiment with a whole new approach to text in the Atari that may be even more convenient. The Atari allows you to treat the keyboard, the video memory, and any other I/O device as a file. I suspect that once I get used to this, I will not really mind giving up PRINT USING.

There is a more definite limitation to the Atari in one of the key text handling areas, and that is in word processing. Forty columns per line is simply not as convenient as the longer lines on some other computers. The problems here are color and expense. It is much easier and cheaper to give text processing ability and sharp resolution to a computer which does not use a video modulator and does not use color. The Heathkit H-89 with 24 lines of 80 characters has much sharper letters than the Atari, yet the Atari limits you to a mere 38 to 40 characters. A lot of this problem could be overcome by designing the Atari to be used only with a high quality color monitor, but that would price it right out of the consumer market. My own solution is to use a different computer for word processing, including the writing of these columns.

Jumps and Subroutines

One of the areas in which Atari Basic enjoys an advantage over Microsoft Basic is in the ability to transfer control to another line through a variable. This has a lot of potential. Look at these comparisons:

```
Atari                 TRS-80
10 GOSUB TIMEOUT      10 GOSUB 500
20 GOSUB BASKET       20 GOSUB 600
10 RATING=            10 RA=INT(BA/5)
  880 + 4 * BASKET
20 GOTO RATING        20 ON RA GOTO 900,
                         920, 940
```

The above example illustrates another difference. In Atari Basic, a variable name may be up to 120 characters long, while the TRS-80 allows only 6 and tests only the first two. In the Atari, VALUE1 and VALUE2 are different variables. In the TRS-80, they are the same.

However, the advantage here is not altogether to the Atari. Radio Shack's Level II Basic allows an ELSE statement, while the Atari does not.

```
Atari                 TRS-80
10 IF A=5 THEN 50     10 IF A=5 THEN 50
20 GOTO 100              ELSE 100
```

Input/Output

A major strength of Atari Basic over Microsoft Basic is in its generalized output routines. This is due to a feature known to mainframe programmers as device orientation. In Atari Basic, PRINT is a generalized output command. While the default device is the video screen, the computer doesn't really care whether it is printing to a line printer, a modem, a cassette tape, a disk file, or the screen. You can even use a variable to shift from one to another in your program, virtually at will. The general format of an OPEN statement hints at the power here:

10 OPEN (Reference number), (input/output/both), extra printer code), device type), device number): (file name). (extension)

Disk file opening might look like this:

10OPEN#2,8,0,“D3:LESSON.BAS”

Chapter 5 in the Atari reference manual gives a more detailed explanation.

What do all these differences mean? My answer is: "Not a whole lot!" Nearly anything that can be done in one Basic can be done in any other Basic, even a limited one like IBM Basic. It just takes extra effort, a little understanding of what the other program is trying to accomplish, and a little creative ingenuity.

An Atari Library of Sound

Richard M. Kruse

Of the recognized human senses, it may easily be argued that the most important are those of sight and hearing. The movie industry was quick to realize the importance of adding sound to their visual productions. First there was simple background music, and later, when it became technically possible, sound was synchronized to the action. Few people today would pay to see a silent movie except under special circumstances.

Yet when most of us think of computers, we usually visualize someone sitting at a video console, typing, and staring silently into the screen. Hollywood generally adds some "bleeps" and "bloops", supposedly electronic, to the background. Real data processing centers are usually quite noisy with machinery running and several printers banging away. These are all artificial sounds, however, far removed from what all of us experience in daily life.

Personal computing, of course, need not follow the same path. If it is technically feasible, why not add the dimension of sound to the already accepted versatility of a good color graphics system? Why not, indeed! Manufacturers of small computers are responding in varying degrees to this challenge. It is now up to programmers to use this new capability effectively.

One of the outstanding features of the Atari 400/800 personal computers is the built-in sound generation system. There is no need to jury-rig an external amplifier and speaker and then operate it with "PEEK s" and "POKE s". Atari's sophisticated sound channels are manipulated through special Basic commands, and the RF output carries the sound information properly formatted to be reproduced through the speaker of a standard television receiver. The television's sound system does not have to be of especially high quality to adequately handle the range of frequencies produced (although it certainly doesn't hurt). An added bonus of this system is that sound and video are presented side-by-side. Most people will probably find this preferable to listening to a disembodied sound source physically separated from the visual presentation.

The Ataris give you not just a single sound generator, but four identical "channels" which may be used separately or in any combination. Each channel has individually controllable pitch and volume, along with a third parameter which Atari calls "tone." The Basic statement which activates one of the sound channels has the following form:

```
100 SOUND P1, P2, P3, P4
```

Parameters P1 through P4 are integer values. P1 specifies which channel is to be activated, identified as zero through three. P2 may be any value from 0 to 255, and sets the relative pitch or frequency of the sound. In the pure tone mode, the pitch range is about two and one-half octaves, and by using a look-up table of conversion factors between musical notes and pitch values, playing a melody on the Atari becomes almost trivial. Playing four-part harmony can be done with some additional programming effort.

One of sixteen different volume levels (including off) is selected by the value of P4.

The tone parameter, P3, is a corker. There are eight possible values, two of which result in relatively pure musical tones. The remaining six, however, are not really "tones" at all, but special effects settings which produce strange and wonderful sounds that will be variously perceived as trucks, helicopters, heavy machinery, and warp drives. These effects, like the pure tones, may be varied in pitch and volume. And always, two or more sound channels may be active simultaneously. As you can see, the number of possible sounds and effects is staggering. Normal sounds can be imitated and new ones created, limited

Richard M. Kruse, Xentrix Engineering, Box 8253, Wichita, KS 67220.

only by the imagination of the programmer.

To stimulate those imaginations, and to show the methods used to put these effects to work, one dozen varied and useful sound effects are presented here. Each effect is programmed as a subroutine which will run for a certain length of time and then terminate. Each subroutine makes use of one or more sound registers, and many of them accept one or more input parameters which modify the effect and/or its running time. A brief explanation is presented for each, so that you will be able to change the effects as desired.

1. Percussive Sound Generator-(See listing 1)

This is a "building block" subroutine which imitates the sound of struck or plucked musical instruments or, with different parameters, explosions or gunshots.

The percussive effect is achieved by executing a loop which initially sets a high volume level, then repeatedly reduces that level by a given percentage until it falls below a present minimum. The volume reduction factor is stored as the variable ICR, and it is easy to see that changing the value of ICR will change the rate of decay of the sound. Since ICR is calculated from the input parameter DUR, the decay rate can be modified at will each time the subroutine is called. The value 10 in statement 10020 is the tone parameter, and results in a pure tone output, so that this subroutine will imitate a chime or bell. Statement 10010 adds a brief burst of white noise at the start of the loop. (It is turned off at step 10025.) This enhances the initial "strike" effect and is heard in the sounds of many musical instruments. Statement 10040 turns the sound off altogether prior to returning to the calling program. While this percussive sound routine will run by itself, it can also be used in the generation of more complex sounds, as will be demonstrated.

2. Doorbell-(See listing 2)

...Now, who could that be?...

The familiar "Dinnng, donnng" of the doorbell is created by two sequential calls to a modified percussive routine. Two different pitches and two moderately long decays are used. What could be simpler?

3. Ringing Telephone-(See listing 3)

...Mildred, would you get that?...

The telephone bell is actually just repeated invocations of the percussive sound, using a high pitch and a short decay. Notice that the two sound registers are set at slightly different pitches. This creates the strident nature of this effect. The final percussive call uses a longer decay time, resulting in a fairly natural "lingering" sound. The apparently meaningless statement at line 10045 simply wastes some time between "rings." You will see this same type of delay in some of the other routines.

4. Alarm Bell-(Seeing listing 4)

...Attention all hands! Secure for hyperwarp...

This is another application of the percussive effect, and is almost identical to the telephone bell. The main differences are that this effect uses a lower pitch and a slower repetition rate. One subtle modification to the percussion routine in both of these effects is the use of a larger value in testing for the end of the decay (notice the variable LM). This is another way to modify the decay time and may be preferable for fast action.

5. Explosion-(Seeing listing 5)

...Hah! Got the little # @ * % !...

The explosion effect is also based on the percussive generator, using "white" (actually "pink") noise instead of a musical tone. For more volume we use three sound registers simultaneously, and to heighten the realism each is given a slightly different pitch. Finally, we use three different rates of decay, the slowest for the lowest pitch. This gives the "rolling" effect of a really "big bang." Entering this subroutine with DUR set to zero will give a pretty fair imitation of a gunshot, since it's basically the same kind of sound.

6. Siren #1-(See listing 6)

...is he after me?...

This routine produces the rising and falling wail characteristic of electromechanicalfire and police sirens. The inner loop in this subroutine (steps 10020 to 10035) generates either an increasing or decreasing pitch of constant amplitude. Each execution of the outer loop (steps 10015 to 10045) reverses the start, stop, and increment values. The delay is used again at step 10030 to waste a little time so that each execution of the loop takes about a second.

7. Siren #2-(See listing 7)

...Quickly, Henri! The Gendarmes...

This alternate siren effect, which I tend to think of as "European," is becoming more common in this country as well, as police and fire departments switch to purely electronic noisemakers. It is one of the simplest effects to create, requiring only alternating high and low pitches at constant volume. The wait loop is used again, at step 10025.

8. Ticking Clock-(See listing 8)

...You have ten seconds to guess the correct answer...

The ticking of a clock (or bomb, heaven forbid) can be nicely simulated by repeated short bursts of white noise. Tone value eight, at a high pitch, serves this purpose. To get a tick-tock effect, two alternating values are used for the pitch parameter.

9. Klaxon-(See listing 9)

...RED ALERT! RED ALERT! Enemy sighted at...

Here, sound registers zero and one operate at slightly different pitches to generate a loud and strident blast, with sound register two filling in a buzzing effect. To add to the realism, one sound register is used at the beginning and end to build up to and decay from the main tone.

10. Whistle and Bomb-(See listing 10)

...Hit the deck!...

For this effect, the percussive explosion of example five is preceded by a convincing anticipatory whistle. Steps 10010 through 10030 create the whistle, which decreases in pitch while increasing in volume.

11. Steam Whistle-(See listing 11)

...All aboarrrrrd! Next Stop Pottsville...

A small amount of white noise from sound register zero in step 10025 adds a realistic hiss to this whistle variation. As in the Klaxon effect, there is a brief build-up preceding the main sound, and a decay at the end.

12. Sawing Wood-(See listing 12)

...And now for something completely different...

This final effect, unrelated to the others, is an example of picking a sound at random and trying to imitate it on the Atari. For sawing wood, you need a buzzing sound... Subroutine 10065. You need to make it rise and fall in pitch as the blade moves...subroutine 10030. For better realism, you need two different pitches as the blade is pushed forward on the cutting stroke and then returned... statements 10015 and 10020.

It is hoped that these relatively simple examples will provide the motivation for Atari owners to get the most out of one of the built-in features of their computers. Other possible effects might include animal imitations, automobile sounds, factory noises, and on and on...the list of possibilities is truly unbounded.

If you have been programming without sound, you will be amazed at the improvement to be gained by its use in games and audio-visual presentations. Once you grow accustomed to this added dimension, it is certain that you will no longer be satisfied with a dull, mute computer.

The secret to success of the small personal computer lies in your creativity and imagination. Put them to work with Atari sound and see what develops. You can't go wrong!

LISTING 1: PERCUSSIVE SOUND GENERATOR

```
10000      REM PERCUSSIVE SOUND GEN
10002      REM ENTER W/2 PARAMETERS
10004      REM NTE=PITCH, 0-255
10006      REM DUR=LNGTH OF EFFECT, 0-10
10010      SOUND 1,5,8,6
10015      VOL=15: ICR=0.79+DUR/50
10020      SOUND 0,NTE,10,VOL
10025      SOUND 1,0,0,0
10030      VOL=VOL*ICR
10035      IF VOL>1 THEN 10020
10040      SOUND 0,0,0,0: RETURN
```

LISTING 2: DOORBELL

```
10000      REM DOORBELL
10002      REM NO ENTRY PARAMETERS
10010      NTE=105: DUR=7.5: GOSUB 10025
10015      NTE=132: DUR=8.5: GOSUB 10025
10020      SOUND 0,0,0,0: RETURN
10025      VOL=15: ICR=0.79+DUR/50
10030      SOUND 0,NTE,10,VOL
10035      VOL=VOL*ICR
10040      IF VOL>1 THEN 10030
10045      RETURN
```

LISTING 3: TELEPHONE BELL

```
10000      REM TELEPHONE BELL
10002      REM ONE ENTRY PARAMETER
10004      REM TMS=# RINGS
10010      FOR XX=1 TO 35
10015      IR=0.3: LM=2: GOSUB 10055
10020      NEXT XX
10025      IR=0.9: LM=1: GOSUB 10055
10030      SOUND 0,0,0,0: SOUND 1,0,0,0
10035      TMS=TMS-1
10040      IF TMS<1 THEN RETURN
10045      FOR WT=1 TO 300: NEXT WT
10050      GOTO 10010
10055      VL=15
10060      SOUND 0,40,10,VL
10065      SOUND 1,42,10,VL
10070      VL=VL*IR
10075      IF VL>LM THEN 10060
10080      RETURN
```

LISTING 4: ALARM BELL

```
10000      REM ALARM BELL
10002      REM ONE ENTRY PARAMETER
10004      REM DUR=APPROX SECONDS RUN
10010      FOR TMS=1 TO DUR*12
10015      VL=15: IR=0.5: LM=3: GOSUB 10040
10020      NEXT TMS
10025      VL=10: IR=0.95: LM=1: GOSUB 10040
10030      SOUND 0,0,0,0: SOUND 1,0,0,0
10035      RETURN
10040      SOUND 0,53,10,VL: SOUND 1,60,10,VL
10045      VL=VL*IR
10050      IF VL>LM THEN 10040
10060      RETURN
```

LISTING 5: EXPLOSION

```
10000      REM EXPLOSION
10002      REM ONE ENTRY PARAMETER
10004      REM DUR=LNGTH OF EFFECT, 0-10
10010      NTE=20: GOSUB 10025
10015      SOUND 1,0,0,0: SOUND 2,0,0,0
10020      RETURN
10025      SOUND 2,75,8,15
10030      ICR=0.79+DUR/100
10035      V1=15: V2=15: V3=15
10040      SOUND 0,NTE,8,V1
10045      SOUND 1,NTE+20,8,V2
10050      SOUND 2,NTE+50,8,V3
10055      V1=V1*ICR
10060      V2=V2*(ICR+0.05)
10065      V3=V3*(ICR+0.08)
10070      IF V3>1 THEN 10040
10075      SOUND 0,0,0,0: RETURN
```

LISTING 6: AMERICAN SIREN

```
10000      REM SIREN 1
10002      REM ONE ENTRY PARAMETER
10004      REM DUR=APPROX SECONDS RUN
10010      LO=50: HI=35: STP=-1
10015      FOR TIM=1 TO DUR
10020      FOR NTE=LO TO HI STEP STP
10025      SOUND 0,NTE,10,14
10030      FOR WT=1 TO 20: NEXT WT
10035      NEXT NTE
10040      XX=LO: LO=HI: HI=XX: STP=-STP
10045      NEXT TIM
10050      SOUND 0,0,0,0: RETURN
```

LISTING 7: EUROPEAN SIREN

```
10000      REM SIREN 2
10002      REM ONE ENTRY PARAMETER
10004      REM DUR=APPROX SECONDS RUN
10010      LO=57: HI=45: NT=HI
10015      FOR TIM=0 TO DUR*2
10020      SOUND 0,NT,10,14
10025      FOR WT=1 TO 180: NEXT WT
10030      NT=LO: LO=HI: HI=NT
10035      NEXT TIM
10040      SOUND 0,0,0,0: RETURN
```

LISTING 8: TICKING CLOCK

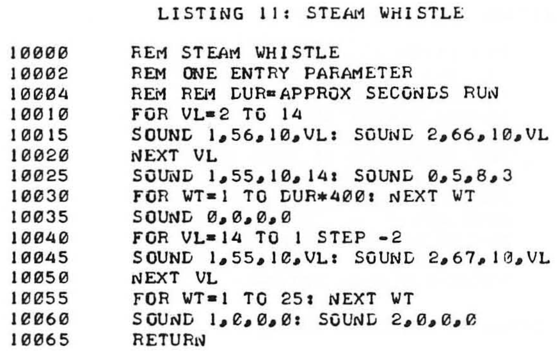
```
10000      REM TICKING CLOCK
10002      REM TWO ENTRY PARAMETERS
10004      REM SIZ=1(FST) TO 10(SLW)
10006      REM DUR=APPROX SECONDS AT SIZ 5
10010      TIC=SIZ+5: TOC=SIZ+10
10015      FOR TIM=1 TO DUR
10020      SOUND 0,TIC,8,12: GOSUB 10035
10025      SOUND 0,TOC,8,12: GOSUB 10035
10030      NEXT TIM: RETURN
10035      SOUND 0,0,0,0
10040      FOR WT=1 TO SIZ*34: NEXT WT
10045      RETURN
```

LISTING 9: KLAXON

```
10000      REM KLAXON
10002      REM ONE ENTRY PARAMETER
10004      REM DUR=APPROX SECONDS RUN
10010      FOR TIM=1 TO DUR
10015      FOR NT=1 TO 10
10020      SOUND 0,100-NT,10,10
10025      NEXT NT
10030      SOUND 0,90,10,14
10035      SOUND 1,95,10,12
10040      SOUND 2,20,2,4
10045      FOR WT=1 TO 200: NEXT WT
10050      SOUND 1,0,0,0: SOUND 2,0,0,0
10055      FOR NT=1 TO 5
10060      SOUND 0,90+NT,10,8
10065      NEXT NT: SOUND 0,0,0,0
10070      FOR WT=1 TO 100: NEXT WT
10075      NEXT TIM: RETURN
```

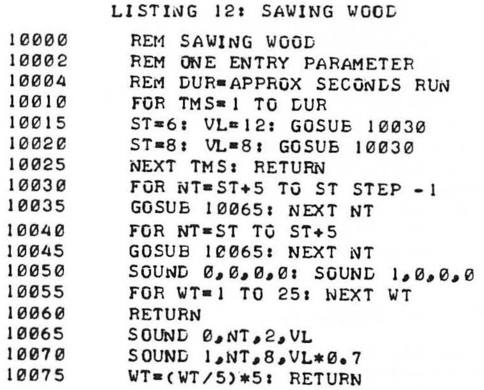
LISTING 10: WHISTLE AND BOMB

```
10000      REM WHISTLE & BOMB
10002      REM ONE ENTRY PARAMETER
10004      REM DUR=LNGTH OF EFFECT
10010      V1=4: FOR NT=30 TO 75
10015      SOUND 0,NT,10,V1
10020      SOUND 1,NT+3,10,V1*0.7
10025      FOR WT=1 TO DUR*3: NEXT WT
10030      V1=V1*1.03: NEXT NT
10035      SOUND 2,35,8,12
10040      V1=15: V2=15: V3=15
10045      NT=DUR+5: ICR=0.79+DUR/100
10050      SOUND 0,NT,8,V1
10055      SOUND 1,NT+20,8,V2
10060      SOUND 2,NT+50,8,V3
10065      V1=V1*ICR
10070      V2=V2*(ICR+0.05)
10075      V3=V3*(ICR+0.08)
10080      IF V3>1 THEN 10050
10085      SOUND 0,0,0,0: SOUND 1,0,0,0
10090      SOUND 2,0,0,0: RETURN
```



```
              LISTING 11: STEAM WHISTLE

10000       REM STEAM WHISTLE
10002       REM ONE ENTRY PARAMETER
10004       REM REM DUR=APPROX SECONDS RUN
10010       FOR VL=2 TO 14
10015       SOUND 1,56,10,VL: SOUND 2,66,10,VL
10020       NEXT VL
10025       SOUND 1,55,10,14: SOUND 0,5,8,3
10030       FOR WT=1 TO DUR*400: NEXT WT
10035       SOUND 0,0,0,0
10040       FOR VL=14 TO 1 STEP -2
10045       SOUND 1,55,10,VL: SOUND 2,67,10,VL
10050       NEXT VL
10055       FOR WT=1 TO 25: NEXT WT
10060       SOUND 1,0,0,0: SOUND 2,0,0,0
10065       RETURN
```



```
           LISTING 12: SAWING WOOD
10000        REM SAWING WOOD
10002        REM ONE ENTRY PARAMETER
10004        REM DUR=APPROX SECONDS RUN
10010        FOR TMS=1 TO DUR
10015        ST=6: VL=12: GOSUB 10030
10020        ST=8: VL=8: GOSUB 10030
10025        NEXT TMS: RETURN
10030        FOR NT=ST+5 TO ST STEP -1
10035        GOSUB 10065: NEXT NT
10040        FOR NT=ST TO ST+5
10045        GOSUB 10065: NEXT NT
10050        SOUND 0,0,0,0: SOUND 1,0,0,0
10055        FOR WT=1 TO 25: NEXT WT
10060        RETURN
10065        SOUND 0,NT,2,VL
10070        SOUND 1,NT,8,VL*0.7
10075        WT=(WT/5)*5: RETURN
```

Ram Cram Techniques for Atari

Original Adventure in 32K

Robert A. Howell

A few months ago something new was added to my family. A 10-lb. 16" by 12" by 4" Atari 800 computer. Not only that, this new computer had no disk. That's right, no disk. Only a cassette recorder to save and load programs and 32K (32,768) bytes of RAM. After having spent 17 years of my life talking to big computers with million of bytes of memory and unlimited disk space (well, almost unlimited), I was understandably a little nervous about the usefulness of such a small computer.

About this same time, I had just finished several weeks of lunch hours (half hours if my boss is reading this) doing some fantastic arm chair spelunking. Yes, I had become hooked on exploring that colossal underground cave where magic is said to work and others had found fortunes in treasure and gold!

My large, friendly computer at work had been my eyes and hands guiding me past giant snake and dragon through scores of rooms deep underground. I even tricked a troll. I was able to retrieve 15 magnificent treasures bringing them to the surface to be mine forever! Once in that cave it wouldn't let me give up, as I soon discovered, until finally, finally, many lost lunch hours (half hours if my boss is still reading this) later, every corner and dead end had been explored, a map of the cave was in hand and I had solved the original "Adventure."

Then a thought came to mind. I promptly dismissed it as absurd. But the thought kept haunting me until it became a challenge. Could this tiny little 32K computer with no disk which I now owned—could it possibly handle "Adventure"? Would the original Crowther and Woods Adventure program fit into 32,768 bytes of memory? I had seen several versions of this program advertised which required at least one disk drive and 32K or more of memory, but none for my little one. Was my little computer really equal to the task, or was I just fooling myself?

The challenge lay before me: get "Adventure" running in Basic on an Atari 800 computer with no disk and only 32K bytes of memory. Little did I realize what I was getting myself into when I accepted this challenge. A challenge that would certainly tell me if this new little addition to the family was really a giant in disguise!

Have you ever spent your whole summer beside the swimming pool out back, with

Robert A. Howell, 20 Richman Road, Hudson, NH 03051.

the tops of your hands, shoulders and knees burning up from the sun, never once getting your swimming suit wet? No? Well then, you have never spent the summer trying to cram "Adventure—messages and all—into 32K of RAM. I did. And to spare you the gruesome details, suffice it to say that I accepted the challenge and won! Just as it was time to close down the pool for the winter, "Adventure" was running on my big computer (never again to be called "little").

The messages and vocabulary were not as extensive as in the original, but they were there, along with the rooms in various colors (except the "all alike" maze where passages and dead ends were all black). Almost everything from the original "Adventure" was included.

Now I know what you just said. You said, "How did he do it?" Well if you didn't say that then you should have, because that's the purpose of this article. As a result of my programming effort, as well as missing out on a whole season of swimming, I learned many techniques for efficient use of memory in Atari Basic. I am going to pass these along so that you will never need to worry about the swimming season passing you by.

Although my examples and techniques refer to Apple Basic and "Adventure" type programs in particular, most of them can be applied to any computer and to programming in general. Why purchase 48K of memory and two disk drives, when in many instances 32K or less of memory is all you really need. Why bemoan the fact that the latest "GLOP" game from the pages of this magazine requires 10K of RAM and your computer only has 8K. Apply a couple of the techniques that I am about to describe and you can probably get the program into 7K or less without losing a single feature!

REMarks

Although I realize that adequate documentation is often lacking in many programs today, when memory is at a premium, REMark statements must be sacrificed. A remark N characters long including imbedded blanks) occupies N+3 bytes if on the same line as another statement and N+6 bytes on a line by itself. Thus, REM's interspersed throughout a large program can waste a significant amount of memory.

An alternative which I use successfully, is to keep the remarks separately on paper, refering to the line numbers in the program. As the program is developed and changed, these remarks are also updated. Then, when the program is finished, a good set of documentation is already available. Also, by maintaining an up-to-date set of remarks, I found I was able to debug the program much more quickly. I estimate I saved about 1000 bytes of memory by eliminating the REMark statements from my "Adventure."

Line Numbers

When a new line (with a new line number) is added to a program, 6 bytes of memory are required by the new line. When that same Basic statement is added to an already existing line, only 3 additional bytes are required. Therefore, 3 bytes of memory are saved each time a new statement is added to a line which already exists. (Multiple statements per line are, of course, separated by colons.) To illustrate the savings that can result, in my version of "Adventure" there are about 720 individual Basic statements (not including DATA statements) but only 325 line numbers. This saves (720-325)*3 or 1185 bytes of memory.

Having written programs for many years using one statement per line, I was a little apprehensive about how difficult multiple statements per line would make program legibility and debugging. However, I found I had no trouble whatsoever reading the program and working with it, even though the Basic statements were packed very tightly.

Putting more than one statement on a line can cause problems if one is not careful, especially in a Basic that contains no ELSE capability. Consider the following example:

```
100 SUM=0
110 FOR I=1 TO 10
120 IF A(I)>0 THEN SUM=SUM+A
  (I)

130 NEXT I
140 PRINT SUM
```

One would be tempted to rewrite this sequence all on one line (with one line number) as follows:

```
  100 SUM=0: FOR I=1 To 10: IF
A(I)>0 THEN SUM=SUM+A(I):
  NEXT I: PRINT SUM
```

However, this puts the NEXT and PRINT statements under the control of the IF, causing them to be executed only when the IF is true. This will produce incorrect results. The proper way to consolidate these statements is:

```
  100 SUM=0: FOR I=1 TO 10: IF
A(I)>0 THEN SUM=SUM+A(I)
  130 NEXT I: PRINT SUM
```

Statements after an IF should be placed at the front of the following statement, or on a line by themselves if the following statement has a branch to it from elsewhere in the program. Of course if the statements after the IF clause are to be executed only when the IF condition is true, then they must be left on the same line as the IF statement.

Here is another example which sets X to 10 or 20 depending on the value of L:

```
100 IF L=R THEN 400
200 X=10
300 GOTO 1000
400 X=20
500 GOTO 1000
```

This section of the program can be neatly condensed into two lines as follows (eliminating one GOTO and saving 27 bytes):

```
100 X=10: IF L=R THEN X=20
200 GOTO 1000
```

Again, the GOTO 1000 must be placed on a separate line so it does not fall under the control of the IF statement.

It may not be obvious what will happen when some statements in Atari Basic are imbedded in the middle of a multi-statement line. Figure 1 lists these statements with an explanation of what happens to statements which follow each of them on the same line.

Make a similar table for your Basic by trying out each statement in a small test program. Then keep this table handy for reference when you are optimizing a large program.

Another way to eliminate line numbers is by inserting a NOT in front of an IF condition. For example:

```
100 IF A=1 AND B>5 THEN 130
110 B=B-1
120 GOTO 1000
130 PRINT
```

may be rewritten on two lines (saving 11 bytes) as follows:

```
  100 IF NOT(A=1 AND B>5) THEN
B=B-1: GOTO 1000
  130 PRINT
```

Here is a different example that may occur in a program:

```
  90 ON X GOTO 100,200,300,400
  100 T=0: GOTO 1000
```

These two lines may be condensed onto one line:

```
  90 ON X-1 GOTO 200,300,400: T=0:
GOTO 1000
```

eliminating line 100 and saving three bytes.

There are many other ways that multiple statements may be squeezed onto one line in order to save memory. A program that does not already do this can probably be reduced to 75% or 50% of its original line numbers. Keep in mind however, that a branch to a statement from elsewhere in the program requires that statement to be at the beginning of a line. Also, in Atari Basic this technique is limited by the length of a logical line which is equal to a maximum of three physical lines or 120 characters. Greater savings can be obtained using Basics which allow more characters per logical line.

Don't Use Constants!

One of the biggest memory wasters in Atari Basic is the use of constants. Each occurrence of a numeric constant or a line number in a Basic statement is replaced by one byte pointing to the memory location where the value of that constant is stored. This value in memory is stored in internal binary form and occupies an additional 6 bytes regardless of the size of the constant. Therefore, each use of a numeric constant or line number in a statement requires 7 bytes of memory. This method of storing numeric constants is what would be expected. Now for the bad news. Since Basic is an interpreter (that is, every statement is kept in memory in almost its original form and decoded each time the statement is executed), when it encounters a constant in a new statement being entered in, it has no way of knowing if that constant was used before. Therefore, it just goes ahead and converts into internal binary form and stores it in memory again using another 7 bytes.

Now, suppose a large program uses the constant 0 (zero) 50 times. Then that one constant occupies 7 times 50 or 350 bytes of memory! Likewise, suppose line number 100 is referenced in GOTO and IF...THEN statements 50 times throughout a program. That one line number also occupies 350 bytes of memory. So we have 700 bytes of memory being used to store the two values 0 and 100. Wouldn't it be nice if each new use of the same constant or line number would point to the memory locations where that value was stored the first time?

Fortunately, there is a way to make that happen: by the use of variables in place of numeric constants and line numbers. The first time a variable is used in a statement four things happen:

1. The variable name is placed in a table in memory called the VNT (Variable Name Table).

2. Six bytes of memory are allocated to store the value of the variable.

3. Two additional bytes are stored in the VNT which point to the value of the variable in memory.

4. One byte is placed in the Basic statement in place of the variable name. This byte points to the VNT.

Thus N+6+2+1 or N+9 bytes of memory are used to store the first occurrence of a variable name (where N is the number of characters in the name of the variable). Now the memory-saving aspect of this method comes into play with the second, third, etc. time the variable name is used. Each subsequent use causes only 1 additional byte of memory to be allocated: the byte in the Basic statement that points to the VNT. Unlike when a constant is used, the 6 bytes of memory to store the value is not allocated over and over again.

To use this method of replacing constants with variables, one other item must be considered. The variable being used must be initialized with the value of the constant it represents. The most efficient way to do this is with READ and DATA statements at the beginning of the program. In an initialization section, values are read in for all the variables which are being used to replace numeric constants and line numbers.

A good rule of thumb to use in deciding whether or not to replace a particular numeric constant or line number with a variable is the following: If the same numeric constant or line number is used four or more times in a program, memory will be saved by converting it to a variable. If used three or fewer times, leave it in its original form.

Of course, the more characters there are in the variable name and in the constant, the more memory will be used in the VNT (to store the variable name) and in the READ/DATA statements. However, the break between three and four occurrences seems to work in most cases.

Now you are probably saying to yourself, "How can I possibly make any sense out of my program if I convert all the constants and line numbers to variable names?" And I agree. If you can't distinguish between the constants and actual variables, then reading the program listing becomes difficult.

Therefore, decide on a pattern for variable names which will be used to represent numeric constants and line numbers in the program and stick to this pattern. An example of what I use is found in Figure 2.

Then for real variables which do actually vary, I use the names J through Y and names that contain all letters (such as AA, AB, QX, ZZ, etc.). This way I can always distinguish constants from variables. If the program uses negative and decimal constants, then establish a pattern for them also.

Statement	Statements following on same line
DATA	Never executed
DIM	Always executed
END	Never executed
FOR	Always executed
GOSUB	Executed upon RETURN
GOTO	Never executed
IF . . . THEN	Executed on condition true Never executed on condition false
LIST	Always executed
NEXT	Executed when FOR loop is finished
ON aexp GOTO lineno-list	Executed if aexp is less than 1 or greater than the number of line numbers in the lineno-list
ON aexp GOSUB lineno-list	Executed if aexp is less than 1 or greater than the number of line numbers in the lineno-list, otherwise executed upon return from the subroutine
POP	Always executed
REM	Never executed—treated as part of the REMark
RETURN	Never executed
RUN	Never executed
STOP	Never executed
TRAP	Always executed

Figure 1.

Figure 3 is example of a program segment before and after the constant-to-variable surgery has taken place.

Note, when a statement number on an IF...THEN is changed to a variable, a GOTO must be inserted (see line 60 in Figure 3). Other than this one exception, any place a numeric constant or line number can be used in an Atari Basic statement, a variable can be substituted.

Constants	Variable Names
0	Z
1 to 9	A to I
10 to 19	A0 to A9
20 to 29	B0 to B9
.	.
.	.
.	.
90 to 99	I0 to I9
100 to 109	A00 to A09
.	.
.	.
.	.
etc.	etc.

Figure 2.

Also note line 40; even the dimensions in an array can be made variables, thus saving the memory that would be used to store the constant dimensions.

Is it really worth the trouble to convert most of the constants and line numbers in a program into variables? In my "Adventure" program, I changed 58 constants and line numbers to variables and saved over 3500 bytes! This represents 12% of the free memory on a 32K Atari system, so the effort certainly paid off. The maximum number of variable names allowed in a single program in Atari Basic is 128. This is as big as the VNT can get.

Therefore, start with the numeric constants and line numbers that are used most often since these will result in the greatest savings. Also, instead of converting constants which are not used very often, consider that GOTO 9 can be changed to GOTO D+E. This will save changing the constant 9 into a variable if D and E are already defined to be 4 and 5 respectively. The constant 9 requires 7 bytes whereas D+E requires only 3, a saving of 4 bytes of memory. Use this technique of combining variables to replace constants that occur three or fewer times in a program.

As can be seen, substitution of variables for oft-used numeric constants and line numbers can result in a substantial increase in memory available in a program.

Numeric Arrays

How much memory will the following statement use:

```
10 DIM A(100), B(100), C(100), D(100),
E(100)
```

If your answer is 500 bytes, you are not even close. The above dimension statement will require over 3000 bytes of memory. Yes, 3000! Why? As we discussed earlier, numbers in Atari Basic are kept in memory in internal binary form occupying 6 bytes each. Therefore, each of the above arrays uses 100 times 7 bytes of memory apiece, and 5 of them will take 100 times 6 times 5 or 3000 bytes. When the memory space is tight, there are two rules to observe in using numeric arrays: 1. Keep their dimensions as small as possible. 2. Eliminate them whenever possible.

There are several ways to eliminate numeric arrays. I will mention two of them: Convert them to strings, and store numeric data in DATA statements and access it with READ statements each time the data is required.

In Atari Basic, strings must be dimensioned. In the statement:

```
10 DIM R$(100), R(100)
```

R as we now know occupies 600 bytes, but R$ occupies only 100 because it is a string 100 characters long. Now suppose in an "Adventure" program there are 100 rooms and the program keeps track of which rooms have been visited and which have not. Each element of R(100) would represent a room. R would be initialized to all zeros and when a room was entered, the corresponding element of R would be set to 1. Since each element of R holds only a 0 or 1, this same function can be accomplished with string R$(100) using approximately one-sixth the memory. First R$ would be initialized to all "N" characters (representing "No, the room has not been entered") as follows:

```
100 FOR I=1 TO 100: R$(I,I)="N":
NEXT I
```

(Note in Atari Basic, R$(i,j) represents the substring from R$ starting with character i and ending with character j. Therefore, R$(i,i) represents the ith character of string R$.) Then when room number I is entered, R$(I,I) would be set to "Y" (indicating "Yes, the room has been entered"). At the end of the game, the number of rooms visited would be counted as follows:

```
1000 SUM=0: FOR I=1 TO 100: IF
R$(I,I)="Y" THEN SUM=SUM+1
1010 NEXT I
```

Of course, if a numeric array is needed to store many different values, this method will not work. However, for storing just a few different values, try using a string instead of a numeric array and substitute different characters for the various values in order to save on memory.

Now suppose a program uses numeric data that never changes. The room move table in "Adventure" is a good example of this. My "Adventure" has 126 rooms and there are 10 possible directions to take out of each room (N=1, NE=2, E=3, ... NW=8, UP=9, DOWN=10). If an array were used to hold this data, it would contain 126 times 10 or 1260 elements. At 6 bytes for each element, this table would occupy 7560 bytes or almost one-fourth of my 32K memory. The data in this array would be room numbers to move into from each room. So for example, to move West (direction 7) from room 29, the contents of array element (29,7) would be the room number to move into going in that direction. Zero of course would mean no path that way.

This data never changes. Therefore it can be put into DATA statements, one DATA statement per room, 10 numbers (corresponding to the 10 directions) per DATA statement. Suppose the DATA statement for room number 1 is on line 10001, room 2 on line 10002, etc. Also suppose variable DR contains the direction in which the adventurer wishes to go and RC the number of the room he is currently in. Here is how the program would locate the room number to move into:

```
100 RESTORE 10000+RC: FOR J=1
TO DR: READ RN: NEXT J
```

Before	After
	10 READ A,A0,A00,B50,F,I0,Z
	20 DATA 1,10,100,250,6,90,0
40 DIM COUNT (100)	40 DIM COUNT (A00)
50 FOR J=1 TO 100	50 FOR J=A TO A00
60 IF INT(RND(0)*10)+1>6 THEN 90	60 IF INT(RND(Z)*A0)+A>F THEN GOTO I0
70 GOSUB 250	70 GOSUB B50
80 COUNT(J)=COUNT(J)+1	80 COUNT(J)=COUNT(J)+A
90 NEXT J	90 NEXT J

Figure 3.

The RESTORE locates the DATA statement for room RC, the FOR loop reads until the room number corresponding to direction DR is read at the end of the loop. RN contains the desired room number. Using this technique, I saved about 3650 bytes of memory on the room move table in my "Adventure" program.

To go even one step further, I put the data for rooms 1, 2 and 3 all on DATA statement 10003; rooms 4, 5 and 6 on DATA statement 10006 and so forth, thus eliminating two thirds of the DATA statements and saving another 600 bytes. The RESTORE statement will still work in Atari Basic because a RESTORE to line 10001 (for room 1) will actually start reading at line 10003 if lines 10001 and 10002 do not exist. Of course, the FOR loop had to be modifed to read the correct set of 10 room numbers as now there were 30 room numbers per DATA line. With this modification, the room move table has now been reduced from 7560 to about 3300 bytes for a 56% reduction in memory used.

Furthermore, upon examining the room move table data, I found that it contained many zeros. This occurs because there are exits from most rooms in only a few of the 10 directions. Therefore, I replaced n consecutive zeros in the DATA statemetns with the number -n. For example, if one of the DATA statements contained 8 zeros in a row, these zeros were eliminated and a single -8 put in their place. This was done in all DATA statements where 2 or more zeros occurred together. The read routine was then modified to expand negative numbers back to the original number of zeros as the data was read. This modification further reduced the room table from 3300 bytes to 2236 bytes now occupying 70% less space than if a 126 by 10 numeric array had been used. Thus, over 5300 bytes of memory were saved with several very simple modifications to the room move table part of the program.

Since numeric data items require 6 bytes each when stored in numeric variables or arrays, if the data does not change during program execution, keep it stored in DATA statements and READ it when it is needed. Pack it on the DATA statements as tight as you can. Otherwise use string arrays if possible. The fewer numeric arrays a program uses, the more memory will be available to it.

Strings

Although strings require less memory than numeric arrays, still try to keep their length to a minimum. Don't set up A$(100) when the maximum length A$ will ever be is 50 characters. Also, eliminate string variables when possible. If three strings are defined in a program and one of the strings could do the functions of all three, eliminate two of them.

"Adventure" type programs always have a vocabulary of words which they recognize (NORTH, TAKE, DRAGON, INVENTORY, DIAMONDS, etc.). Cut these words down to their first five characters (NORTH, TAKE, DRAGO, INVEN, DIAMO, etc.) Although some games use the first four or three characters, five is about the minimum length which can be used and still make the words unique. When the player's input is received, each word of it is truncated to five characters before a search is done against the vocabulary in the program.

As discussed previously with numeric data, place the vocabulary in DATA statements and READ it when it is required. In Atari Basic, strings are placed in DATA statements without quotes and are separated by commas. Since Atari Basic does not have string arrays (e.g. A$(100) does not mean 100 strings, but defines a string to be a maximum of 100 characters long), to store the words otherwise, they would need to be packed into a string. Since the words are variable in length (INVEN is five characters long but TAKE is four, OIL, three, etc.), this would require extra program statements and overhead. With the vocabulary on DATA statements, it may be searched by READing it from beginning to end with a special character (*, $, etc.) marking the end of the table.

This will take a considerable amount of time, especially for words at the end of the table. Therefore, a more efficient way is to place all words beginning with the same letter in a separate DATA statement. Then a RESTORE is used, keyed off the first character of the word being searched for, to locate the DATA statement containing all words starting with this letter.

As can be seen, putting both numeric and string data into DATA statements can be a very effective way to reduce the amount of memory required by a program. Before numeric arrays and strings are set up, consider the use of the READ/DATA statement technique. It may make the difference in being able to get a program into memory.

Eliminate Unneeded Statements

When you need to alternate a program variable between 0 and 1, how do you do it? Before reading on, take a piece of paper and write the Basic statements to set B equal to 0 if its value is 1 and vice versa (keeping in mind that Atari Basic does not have an ELSE capability). Now look at your programming. Is this the way you did it?

```
10 IF B=0 THEN B=1: GOTO 30
20 B=0
```

Or how about this way?

```
10 ON B+1 GOTO 20: B=0: GOTO 30
20 B=1
```

Or better yet?

```
10 ON B+1 GOTO 20: B=-1
20 B=B+1
```

Each of these methods is good and will accomplish the task, but they all use two lines. Is there a way (without using ELSE) to write this code on one line? Yes there is. A little creative programming reveals the following method:

```
10 B=ABS(B-1)
```

The first three examples require 52, 60 and 53 bytes respectively; the last example only 20 bytes. The point here is, eliminate unneeded statements wherever possible to save on memory.

I found in my "Adventure" program that the statement:

```
Z=B: GOTO 90
```

occurred 16 times. So I did the obvious: kept the first occurrence of this statement and replaced the other 15 with a branch to the first one. Now I know I have just caused a program abort to occur in the mind of every structured programmer in the audience. Please note, I am not against structured programming.

In fact, I encourage it along with good program documentation wherever possible. However, the preceding example saved 90 bytes of memory. By doing this same thing with several other statements that occurred multiple times in the program, I was able to save another several hundred bytes. So use of this technique really paid off.

Untokenize The Program

When Atari Basic places a statement into memory, it uses a "tokenized" form. That is, each Basic keyword, each arithmetic operator and each relational operator are replaced by a unique 1-byte code called a token. At the same time (as previously discussed), constants are placed in memory in internal form and variable names are placed in the VNT (Variable Name Table). This is the way Basic interpreters work. Thus, they automatically provide some efficiency in their use of memory.

Now, after a new program is entered

into memory, typically a debugging phase begins. The program is run and rerun (and rerun and rerun and rerun....) many times to find and correct as many logic errors as possible. In this phase, statements are added, changed, deleted, rewritten, etc. If the program is large, debugging may take many days or weeks. During this time a number of variable names which were once used in the program will probably be completely eliminated. Or a typing error may have caused the variable TB, for example, to be entered when T was supposed to be used. Later on, this error is discovered and TB is replaced by T in the statement, thus completely eliminating the variable TB from the program.

Sounds logical so far, doesn't it. However, something else occurs that is not immediately obvious. When TB is replaced by T, Basic, being an interpreter, does not know that the variable TB has been completely eliminated from the program. Basic has no way of knowing that TB is now used in any other statement. Therefore, TB still occupies 4 bytes in the VNT and 6 additional bytes of memory are still reserved to hold the value of TB. Ten bytes of memory are being used by TB. Multiply this by another 10 or 15 variables that may have been used in the program but have since been eliminated, and we find a hundred or more bytes of memory being wasted.

"Well," you respond, "when I CSAVE the program onto tape and then CLOAD it back into memory, doesn't the VNT and related memory get cleaned up?"

The answer to this question is "No," because a CSAVE causes the tokenized version of the program to be written onto tape and along with the tokenized program, the VNT and associated memory are also written. One of the reasons CSAVE works this way is because the tokenized version takes much less time and tape to write out. Now when a CLOAD is done, the old VNT still containing the unused variable names and their associated memory is read back in unchanged.

How do you eliminate the unused variables from the VNT and free up their memory bytes? Simple. The program must be written out in its untokenized form. This is the form that is seen on the screen when the program is listed with the LIST command. In Atari Basic, this is done exactly like a CSAVE except the command LIST"C" is used. LIST"C" causes the program to be LISTed to cassette tape. The tape will be written with the untokenized version of the program only and not include the VNT nor any other values from memory.

Note, this process will take two to three times as long as CSAVE and require at least twice as much tape. The tape should then be rewound, NEW typed to clear memory (this is important to erase the old program and VNT), and the untokenized version read back in with the command ENTER"C" (which works just like CLOAD). The untokenized statements will be read in one by one, retokenized and a new VNT constructed. Since the old variable names are no longer in this set of Basic statements on tape, they will not be entered into the new VNT.

When I had finished debugging my "Adventure" program, I untokenized and retokenized it and gained 150 bytes of memory. This allowed me to add a few more vocabulary words that I had previously eliminated for lack of space. Note also that a program should be untokenized and retokenized whenever an ERROR 4 occurs. Error number 4 means the VNT is completely full with 128 variable names. Of course, if the program actually has 128 legitimate different variable names, then this method will not work and some of the variable names must be eliminated or combined into an array (which takes up only one slot in the VNT).

Use of POP Statements

When I finally had the "Adventure" program finished, there were 450 bytes of memory available after loading and 50 bytes free after execution began. I felt the program was now bug-free and ready for the final test: my 10-year old son, David. But a strange thing began to happen. After David played for one or two hours, ERROR 2 would occur and the program would abort. Error number 2 means out of memory. This error would occur randomly after about an hour of play without restarting the game, and always at a different spot. How could this be? I had very carefully calculated that there should be at least 50 spare bytes of memory. I was puzzled. It took me a while to figure out what the problem was, but I finally found it.

When a GOSUB is executed, Atari Basic puts the return address into a push-down, pop-up stack in memory. Then when the RETURN statement is executed, the top address is popped off of the stack and the computer returns control to the program at this address. Thus the stack is constantly expanding and contracting in memory as GOSUB's and RETURN's are executed. Now suppose a subroutine branches elsewhere in the program, never executing a RETURN statement. The return address remains on the stack forever. This is exactly what was happening in my program. Every once in a while the program would exit from a subroutine without executing a RETURN. Each time this happened, 4 bytes of memory remained on the stack, never to be released, and the stack gradually expanded until it had eaten up the 50 bytes of available memory.

There are two ways to eliminate this problem. The most obvious is to exit from every subroutine via a RETURN statement. However, it is not always possible nor desirable to do this. Therefore, before branching out of a subroutine where the RETURN will never be executed, a POP statement should be inserted. This causes the stack to be popped up one time, and the return address removed, just as if the RETURN statement had been executed. The format of this statement is:

```
100 POP
```

In my program, I put several POP statements just before the INPUT statement. The program continually returns here to get the player's next response. Thus, I made sure at this point that the stack was completely empty. Executing a POP when the stack is empty acts like a do-nothing statement and does not cause an abort. This small modification solved the problem.

One note of caution when using POP statements in Atari Basic: FOR loops are also placed on the stack. Therefore, if a program is in the middle of a FOR loop when a POP is executed, the FOR information may be removed from the stack. This will cause the program to abort with error number 13 (NEXT encountered with no matching FOR) when the corresponding NEXT statement is executed. The way to avoid this is to make sure POP statements are not placed within FOR loops, or to make sure that you know exactly what order FOR and GOSUB information was placed on the stack so it may be correctly popped off. Note also that branching out of a FOR loop without completely finishing the loop does not cause the stack to grow and waste memory like GOSUB's do, so one only needs to be concerned about this problem when branching out of subroutines without executing a RETURN.

Message Text

Approximately one half of the memory in my "Adventure" program is text consisting of room descriptions and messages. Since the original "Adventure" text is too large to fit, it had to be cut down. There are several way to do this.

One way is to eliminate completely a number of the least used, least important messages. Another way is to delete some of the descriptive adjectives and/or change the wording so that the message is smaller but still retains its original meaning. Abbreviating, using contractions and substituting smaller words all help considerably. Here is an example:

Original message: "You are in a complex junction. A low hands and knees passage from the north joins a higher crawl from the east to make a walking passage going west. There is also a large room above. The air is damp here."

Abbreviated message: "You're in a complex junction. A low N pass joins a higher crawl from the E making a walking passage W. There's a large room above. The air is damp." Counting spaces, the message has been reduced by 28% from 204 to 147 characters.

Half of the room descriptions (63) begin with the 11 characters 'You are in" (including the space following the word in). I eliminated these words from the front of those 63 messages and modified the print message subroutine to print them if the first character of the message to be printed was not a capital letter. This resulted in another 600-byte savings.

Message text is stored in DATA statements. Message number 1 at line 15010, message 2 at line 15020, etc. The start of a message is located with a RESTORE 15000+N*10 where N is the message number. Many of the messages extend onto multiple DATA statements.

A special character which is not used anywhere else in the message text was placed at the end of every message. This character is detected by the print message subroutine telling it when the end of the message has been reached. Adding this single character per message was the simplest way to allow the program to determine the end of a message when the messages were variable in length. This method also uses the least amount of memory.

With a little creative rewriting, the original "Adventure" message text was cut approximately in half so that it fit into 14K to 15K of memory, but still retained its original meaning. The attractiveness of the game was not lost, and all of the excitement of the original "Adventure" was still there even though the messages were now in an abbreviated form.

Miscellaneous

Here are a few other hints for optimal memory use:

1. Do not use long variable names (Atari Basic allows up to 120 character names, all characters significant). Each character in a variable name occupies 1 byte of memory in the variable name table.

2. Replace IF X <> 0 with IF X (which is equivalent and saves 3 to 9 bytes depending on whether the 0 is a constant or a variable).

3. Use GOSUB's to eliminate multiple occurrences of identical program statements.

4. Use as few variable names as possible by making them do double and triple duty. Rahter than use I, J, K, L, M and N as FOR loop variables, or Z0 through Z9, see if you can get along with using just I and J or Z1 and Z2. The same applies to scratch variables and other variables in the program.

5. Remove unnecessary parentheses and rely on operator precedence wherever possible (except, due to a known bug in Atari Basic, always enclose NOT and its associated variable in parentheses—(NOT B) instead of NOT B).

6. Spaces may be used anywhere for program readability (except of course in strings). Spaces are not stored in memory when a program statement is tokenized.

7. A new line always requires 6 bytes of overhead regardless of the size of the line number used.

8. Change IF NOT (A=B and C=D and E=F...)
to IF A <> B OR C <> D OR E<> F...
Change IF NOT (A=B or C=D or E=F...)
to IF A<> B AND C <> D AND E <> F...
Memory will be saved in both cases.

9. Use of the LET keyword does not cause extra memory to be allocated. It may be included or omitted as desired.

10. The RUN command clears all simple numeric variables to zero and sets all strings to empty (length zero) so don't waste memory clearing them. However, numeric arrays are not cleared! If they must be initialized to zero, use a FOR loop (all on one line, of course).

11. Many of the Atari Basic keywords can be abbreviated. Abbreviations have no effect on memory utilization.

One last question lingers which must be answered: After optimization for efficient memory use with these methods, how slowly does the program actually run? When I had finished the "Adventure" program, I did find the response to the player's input to be too slow. It was in the five to ten second range. However, upon investigation I discovered that the program was taking five seconds searching the vocabulary list. The further down the list it had to search, the longer it took. Therefore, I did sacrifice some memory by placing words that start with the same letter together on separate DATA statements. Then I changed the search routine to do a RESTORE to the proper DATA statement keying off of the first letter of the word that was being searched for. This reduced the search from a maximum of 150 words to 20 or less. Also, I placed the most often used words at the beginning of each DATA statement. Thus the vocabulary is not packed as tightly onto DATA statements as it could be. However, with this one small change, response time is now in the one to two second range for most responses, with a maximum of five seconds for the GET/TAKE verb which has the largest number of program statements associated with it. It appears that, on the Atari 800, chaining constants to variables, reading from DATA statements, jumping all over the place with GOTO's and GOSUB's etc. doesn't cost the program too much in time. This, of course, may not be the case for a program that uses some of the fancy Atari sound and graphics capabilities. However, for "Adventure" in graphics mode 0 (full screen text), the speed is adequate, even when a Basic program is highly optimized for memory usage.

Memory Saving-Techniques

Here is a summary of the memory-saving techniques discussed in the accompanying article:

1. Eliminate REMarks.

2. Pack multiple statements per line to eliminate numbers.

3. Replace constants and line numbers with variables.

4. Reduce the dimensions of and/or eliminate numeric arrays (convert to strings or use DATA statements).

5. Keep strings small and put them on DATA statements.

6. Eliminate all unnecessary statements, especially multiple copies of the same statement.

7. Untokenize and retokenize.

8. Keep the FOR/GOSUB stack from eating up memory.

9. Reduce the size of message text.

10. Use short variable names.

11. Replace IF X <> 0 with IF X.

12. Use subroutines to eliminate duplicate statements.

14. Eliminate unnecessary parentheses.

15. Rewrite to eliminate NOT.

16. Don't initialize to zero exception—numeric arrays).

From Burn-Out to Born Again

Witold Urbanowicz

'Twas the week before Christmas and all through the house...You get the picture. I had just come in from walking the dog. It was late and, more important, the house was quiet. The rest of the family was tucked away for the night. So was the Atari 800.

There it sat neatly stacked on the living room shelf. Next to the cassette player lay operating manuals along with little boxes labelled *Bio-Rhythms* and *Star Raiders*.

Vacationing friends had kindly consented—after a few subtle hints from my nine-year old son—to leave their personal computer in our care. So for the past week, our living room had been an extraterrestrial battle-zone. The walls echoed with the sounds of hyper-space thrusters, photon-lasers and thermonuclear explosions. My wife and I kept our distance. She could not stand the noise and the violence. For me, there were other reasons.

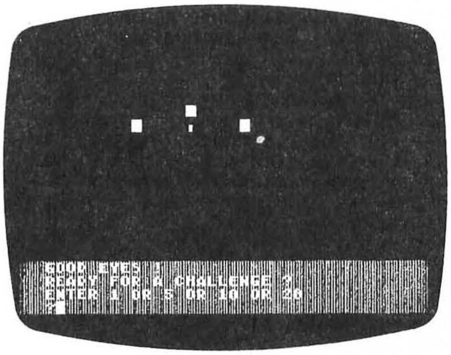
I had spent the better part of my early adult life programming, analyzing, and trouble-shooting commercial computer systems. As I looked at the compact console sitting there on the shelf, my mind went back about ten years.

I could still remember being ushered through a door into a room about the size of a basketball court. It was like walking into the future. Inside this air conditioned, climate controlled world sat the company's four computer systems, along with their respective disk drives, tape drives, printers, and various other peripherals.

I was led past rows of lights flashing on main frame panels. Long lines of tape drives danced back and forth. Disk packs whirled secretively inside their stacked enclosures. Printers spewed out reports by the truckload. Adrenalin began racing through my own system at a speed, I was certain, approaching that of the computers themselves.

Each of the four computer systems had a nickname. "Poppa Bear I" and "Poppa Bear II" were twin systems with individual core capacities in the megabyte range. These were used to run the major financial systems of the corporation. "Momma Bear", weighing in at 512K, was devoted to teleprocessing. It communicated with computers across the country. "Baby Bear", a little 48K three-partition machine was used nights for the smaller systems, but during the day was dedicated to the programming and systems people for compilations and testing. "Baby Bear" was to be my own personal computer for the next few years.

As the years passed, the software became more complex, requiring more and more core as well as faster and faster processing speeds. My skills and responsibilities were naturally upgraded to keep up with the larger and more complex applications. As deadlines became more critical and the problems more intricate, the pace became absolutely frantic. Murphy's Law reigned supreme.



```
GOOD EYES !
READY FOR A CHALLENGE ?
ENTER 1 OR 5 OR 10 OR 20
?
```

"Baby Bear" became a dim memory and my family life was in danger of becoming the same. By now it had become a frequent occurrence for me to work months on end at all hours in order to bring a project in on time. For the most part, the results of this effort—the *bottom line*— were meant for someone I would never meet in a city I would never see.

Several times after the project had been completed, I asked whether it was serving the intended users well, only to discover that the project had been scrapped or that the reports were piling up somewhere in a storeroom in unopened boxes.

About the third or fourth time this happened, I was sitting in the computer room at five in the morning. I had gone about 72 hours without sleep. It had been months since I had seen the kids. I felt as gray and flat as the tile floor beneath my feet. It was then that I decided to leave the field.

When I left, I never wanted to see another computer in my life. Any thrill or satisfaction I may have felt in the early part of my career had not just disappeared. It had been slowly ground out of me.

All these thoughts and images ran through my mind in a matter of nanoseconds that evening as I stared at the Atari on the shelf. It seemed to be waiting for me. I wondered if I had put enough distance between myself and the past. After all I had been out of the business for several years. How could it hurt to sit down for a few minutes to see what the little fellow could do?

In less time than you can say Beginners All Purpose Symbolic Instruction Code, I had the Atari off the shelf and ready to go. Although I had never programmed in Basic before, there was enough of a logical similarity to other high level languages that with a little prompting from the Basic manual I was off and running.

Over the next few evenings, I reintroduced myself to the world of IFs, THENs, GOTOs, strings, subscripts and various other programming concepts. The feeling was definitely odd. It was like meeting old friends I had not seen in years. They had not changed one bit—for better or for worse. Some, like the nested IFs, still caused me no small amount of trouble whenever I took them for granted. Others, as in the case of the GOTO, were still as straightforward as ever.

Then suddenly, about two nights before Christmas as I worked my way through Sound and Graphics, it hit me. Without even realizing it, I was having fun. It was like the old days when I had worked with "Baby Bear" at my first job. But there was a significant difference. The room I sat in was my own. The light by which I worked was soft. The colors and fabrics in the room were part of a human environment. I did not have to fight anyone to get computer time. I was a person enjoying a personal experience.

Now it was time to put together what I had learned the past few evenings. What better way was there than to write a program. The question was, what to write? The answer came quickly. With Christmas not two days away, what better project could there be than to create a present for the children—a small electronic game they could play.

After a bit of thought, I set down some basic specifications. The program would have to be relatively straightforward and short. I didn't have that much time. Plus I was still quite rusty. However, I wanted something that would pose an appropriate challenge to my skills. Finally it also had to be something the children

Witold Urbanowicz, 135 Eastern Parkway, Brooklyn, NY 12238.

would find entertaining as well as challenging.

I finally decided on an electronic version of the old Shell Game. Working into the early hours of the morning, I was able to finish the initial logic for the game, leaving the testing for the following night.

And so the proverbial Night Before Christmas found St. Nick at his Atari testing a last minute present for the big day. Time flew, and soon the morning light seeped into the living room to announce the arrival of the children as they made their way to the tree and the presents which lay beneath. They were somewhat surprised to find me up at that hour.

"What are you doing?" they asked.

"You'll see," I managed to reply.

Paper ripped and the camera clicked. In less than an hour the booty lay displayed. My daughter modeled her clothes while my son booped and beeped his way to electronic heaven with his new hand-held game.

Several hours later, my daughter remembered seeing me when she had first awakened. Leading the kids to the living room, I told them to turn on the computer. This they did and sat down to play. I watched as their faces lit up while trying to follow the shells being moved around on the screen. I followed the squeals of delight when they guessed which shell the pea was under and groaned along with them when their guesses were wrong. Finally my daughter turned to me.

"Daddy, you did this?" she asked.

I nodded.

"Really?" my son added.

All I could do was beam.

Seeing the "bottom line" in my children's faces, I forgot about the hours of frustration the night before while debugging the program. The thrill and satisfaction I had felt years ago when I first worked with "Baby Bear" had returned. Unfortunately, so did the neighbors.

I was actually sad to see the Atari go. Still, I had been given a second chance to look into the future and found once more that it was good. □

Listing 1.

```
5 DIM A$(1)
7 DIM B$(1)
10 MOVE=100:MVA=1:MV2=2:MV3=3:MV4=4
20 MV5=5:PH=0:P1=1:P2=1:P3=4
30 X10=10:X15=15:X20=20:POS1=1
40 NEG1=-1:NXTMOV=600:RNDMOV=800
45 SETUP=900
50 FRTY=40:FIFTY=50
60 Y=10:Z=Y:SV2=2
80 X40=40:X60=60:X80=80
85 ZERO=0:MV10=10
90 GRAPHICS 3
92 COLOR 2
94 SETCOLOR 4,9,2
96 GOTO 300
100 FOR M=MVA TO MVB
110 COLOR 2
120 PLOT X,Y
130 PLOT W,Z
140 FOR S=SVA TO SVB:NEXT S
150 COLOR 4
160 PLOT X,Y
170 PLOT W,Z
180 X=X+XV:Y=Y+YV
190 W=W+WV:Z=Z+ZV
200 NEXT M
205 COLOR 2
210 PLOT X,Y
220 PLOT W,Z
230 RETURN
300 REM * * * MAINLINE * * *
400 GRAPHICS 7
405 COLOR 1
410 SETCOLOR 4,9,2
415 PLOT 61,39:DRAWTO 62,39
420 PLOT 61,38:DRAWTO 62,38
425 Y=32
427 RNDH=ZERO
430 FOR M=MVA TO MV4
435 COLOR 2
440 PLOT X40,Y:DRAWTO X40+3,Y
445 PLOT X60,Y:DRAWTO X60+3,Y
450 PLOT X80,Y:DRAWTO X80+3,Y
455 Y=Y+1
460 NEXT M
465 PRINT "READY?"
470 INPUT A$
475 IF A$<>"Y" THEN 465
480 P1=4:P2=1:P3=4
485 FOR M=MVA TO MV4
490 COLOR 2
500 PLOT X40,Y:DRAWTO X40+3,Y
505 PLOT X60,Y:DRAWTO X60+3,Y
510 PLOT X80,Y:DRAWTO X80+3,Y
515 FOR S=SVA TO FIFTY:NEXT S
520 COLOR 4
525 PLOT X40,Y-4:DRAWTO X40+3,Y-4
530 PLOT X60,Y-4:DRAWTO X60+3,Y-4
535 PLOT X80,Y-4:DRAWTO X80+3,Y-4
540 Y=Y+POS1
545 NEXT M
560 GRAPHICS 3
570 COLOR 2
575 SETCOLOR 4,9,2
580 Y=X10:Z=Y
585 PLOT X10,Y
590 PLOT X15,Y
592 PLOT X20,Y
595 NXTCNT=X15
600 NXTCNT=NXTCNT-POS1
605 SVB=SVB-CH
610 IF NXTCNT=ZERO THEN 650
620 RNDMOV=((INT(3*RND(1)))*10)+800
622 IF RNDH=RNDMOV THEN 620
624 RNDH=RNDMOV
630 GOTO RNDMOV
650 GRAPHICS 7
652 COLOR 2
654 SETCOLOR 4,9,2
656 Y=X20+X20:XA=Y:XB=Y+X20:XC=Y+Y
660 WA=XA+3:WB=XB+3:WC=XC+3
670 FOR M=MVA TO MV4
672 COLOR 2
674 PLOT XA,Y:DRAWTO WA,Y
676 PLOT XB,Y:DRAWTO WB,Y
678 PLOT XC,Y:DRAWTO WC,Y
679 Y=Y-POS1
680 NEXT M
685 PRINT "PICK A SHELL - A,B,C"
687 INPUT A$
690 Y=36:Z=Y
692 IF A$="A" THEN 700
694 IF A$="B" THEN 710
696 IF A$="C" THEN 720
698 GOTO 685
700 C=P1:X=X40:W=X+3
705 GOTO 725
710 C=P2:X=X60:W=X+3
715 GOTO 725
720 C=P3:X=X80:W=X+3
725 FOR M=MVA TO MV2
730 COLOR 2
732 PLOT X,Y:DRAWTO W,Z
734 FOR S=SVA TO FRTY:NEXT S
736 COLOR 4
738 PLOT X,Y+4:DRAWTO W,Z+4
740 COLOR C
744 PLOT X+1,Y+4:DRAWTO X+2,Z+4
746 Y=Y-POS1:Z=Y
748 NEXT M
750 FOR M=MVA TO MV2
752 COLOR 2
754 PLOT X,Y:DRAWTO W,Z
756 FOR S=SVA TO FRTY:NEXT S
757 COLOR 4
758 PLOT X,Y+4:DRAWTO W,Z+4
760 Y=Y-POS1:Z=Y
762 NEXT M
775 IF C=POS1 THEN 790
780 PRINT "PICK ANOTHER"
785 GOTO 687
790 PRINT "GOOD EYES !"
792 PRINT "READY FOR A CHALLENGE ?"
793 PRINT "ENTER 1 OR 5 OR 10 OR 20'
794 INPUT CH:CH=MV2*CH
795 GOTO 300
800 X=X10:W=X20:MV=MV10
803 PH=P1:P1=P3:P3=PH
805 GOTO SETUP
810 X=X10:W=X15:MV=MV5
813 PH=P1:P1=P2:P2=PH
815 GOTO SETUP
820 X=X15:W=X20:MV=MV5
823 PH=P2:P2=P3:P3=PH
825 GOTO SETUP
900 XV=ZERO:YV=POS1
905 WV=ZERO:ZV=NEG1
910 MVB=MV3
915 GOSUB MOVE
920 XV=POS1:YV=ZERO
925 WV=NEG1:ZV=ZERO
930 MVB=MV
935 GOSUB MOVE
940 XV=ZERO:YV=NEG1
945 WV=ZERO:ZV=POS1
950 MVB=MV3
955 GOSUB MOVE
960 GOTO NXTMOV
```

Speedread +

Reading Matters

Bud Stolker

Rising above the glut of "me-too" game programs for the Atari home computers announced at the West Coast Computer Faire last spring is a self-improvement program called *SpeedRead+*. It is a serious attempt to help users boost their reading speed and comprehension by using well designed eye training exercises.

The principle behind *SpeedRead+* is a simple one. If you can train yourself to concentrate on reading text efficiently, you will save time, understand more of what you read, and feel less tired at the end of a long reading session.

To help users achieve these goals, the publisher, Optimized Systems Software, provides a machine language program that flashes words and phrases on the TV screen at speeds from five to five thousand words per minute. Three literary classics are included as text files with the program: Washington Irving's "Rip Van Winkle" and "Legend of Sleepy Hollow," and Bret Harte's "Outcasts of Poker Flat."

Easy to Use

Each eye training exercise is accompanied by tips on how to use it to best advantage. Once an exercise is selected, the user can control both the display speed and the width of the text window by using either the keyboard or the Atari joystick—a nice touch. An option menu and display of the current reading rate are always just a keystroke away.

SpeedRead+ starts by loading the text of your choice into memory, automatically using all the space available. It counts every word of text, so that you can start a session by specifying the very spot at which you left off last time.

The words flash by on the screen, centered under a stationary dot that gives the eyes an anchor in the vast expanse of the video display. The idea is to hold your eyes steady, letting you absorb the information without backtracking or "tuning out." As you feel more confident with the exercise, you can select wider phrase modes (up to 38 characters, nearly the width of the Atari display) to broaden peripheral vision, or you can increase the display speed—or both.

Several Ways to Train the Eyes

From here the alternatives vary, depending on individual needs. A "double phrase mode" displays text alternately on the left and right sides of the screen. This exercise trains your eyes to jump to a predetermined point and instantly recognize the phrases. It also develops the timing and rhythm necessary to read printed text efficiently.

A "random phrase mode" displays text anywhere on the screen, although the user can select the approximate distance from the central stationary dot. The purpose of this mode is to expand peripheral vision, an essential element in speed reading. I found that it took some practice to keep my eyes glued to the dot and still comprehend the text flashing on the screen.

SOFTWARE PROFILE

Name: SpeedRead+

Type: Self-improvement program

System: 16K Atari 400 or 800,
16K Apple II

Format: Disk

Language: Machine

Summary: Useful tool for developing good reading habits and increasing comprehension

Price: $59.95

Manufacturer:
Optimized Systems Software, Inc.
10379 Lansdale Ave.
Cupertino, CA 95014

The "column phrase mode" most closely approximates the kind of reading we all do once we tear ourselves away from the computer. This exercise trains the eyes to travel from top to bottom of a column of text, stopping only once per line and focusing at the center of each line.

The program displays each column for a predetermined number of seconds, then replaces it with more text. By pulling forward or backward on the joystick, I was able to synchronize the speed so that the text changed just as my eyes hit the bottom line of the column.

Method Used In World War II

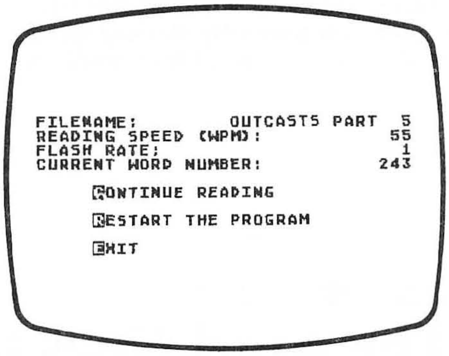
SpeedRead+ is an updated version of the old tachistoscope, a mechanical device that presents visual material for brief periods of time. During World War II, naval aircraft spotters were trained to differentiate friend from foe based on images flashed by tachistoscopes equipped with mechanical shutters. The technique was highly successful. But when they used the machine for character and word recognition, researchers found that average reading rate gains were unimpressive (though some people achieved spectacular gains). To this day the effectiveness of the tachistoscope is an item of controversy. Because *SpeedRead+* is essentially a computerized tachistoscope, it must be evaluated with caution.

Shortcomings

The program makes no attempt to break the text into meaningful phrases; it simply calculates how many words it can display at a time and considers that a phrase. This hinders its ability to boost true phrase recognition and overall comprehension.

While it would have been possible to mark appropriate phrases in the text (by setting the high-order bit of the first character as a flag, for example), this would have entailed a great deal of work, would have raised the cost of the software considerably, and would not have solved the problem of marking user-supplied text.

There are other problems, too. For one thing, the three texts are supplied entirely in upper case. When was the last time you read a book printed in all capital letters? Author Zeissman claims that it is easier to recognize words when they are capitalized. I was taught just the opposite in college design classes. He may be right in the case of the Atari, however. Its lower-case character set is so—well, so *whimsical*—that it could interfere with rapid comprehension.



```
FILENAME:            OUTCASTS PART  5
READING SPEED (WPM):                55
FLASH RATE:                          1
CURRENT WORD NUMBER:               243

       CONTINUE READING

       RESTART THE PROGRAM

       EXIT
```

Intermediate SpeedRead+ menu.

Why Bret Harte?

The choice of texts puzzles me as well. While Bret Harte and Washington Irving are colorful authors, they hardly typify the standard fare of today's readers. Their styles and vocabularies are somewhat dated, and they appeal, I suspect, to a limited audience. I would have preferred to see the *SpeedRead+* manual included on disk so that I could have absorbed it for practice.

Once you have read Harte and Irving a few times, of course, you know every twist in the plots, and I found a tendency to let my mind wander when I should have been concentrating. The author has thoughtfully provided an explanation of

Bud Stolker, Landmark Towers, Apt. 1506, 101 S. Whiting St., Alexandria, VA 22304.

how to create new text files using any Atari-compatible text editor (or the Assembler cartridge). A good way to acquire lots of text is to pull it in over the phone lines from a remote system like the Source or Compuserve.

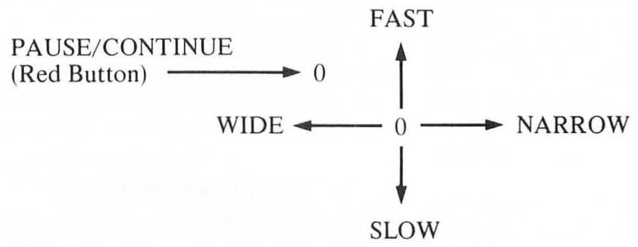
Classroom Use Encouraged

Along with *SpeedRead+* and the text files, OSS includes the framework for an examination program that lets teachers prepare computerized multiple choice tests. The exam system, which includes automatic score keeping, is designed to check students' comprehension levels. Each exam may have up to 255 questions.

The sample test is sketchy indeed, and I couldn't help wishing it had been based on "Outcasts of Poker Flat," a relatively unknown work to many students (and to me). It is probably asking too much of a teacher to make up a computerized exam for each text covered in class, but a school system with several Ataris might use the comprehension exams to advantage.

I suspect *SpeedRead+* will find its way into more homes than classrooms, since it is best used on a regular basis in a quiet place, rather than for a week or two at a time at school.

I found *SpeedRead+* a practical and useful tool. The morning paper has always been my nemesis; a careful reading takes as much as an hour a day. By using *SpeedRead+* as an exerciser, I have raised my comprehension level (though without, alas, cutting my reading time). The improvement may be due to the constant reminders in the manual to concentrate while reading, rather than to any improvement in eye movement or phrase recognition. At any rate (pun intended), I am enjoying my reading more now, and I am convinced that *SpeedRead+* has contributed to my pleasure.



Keyboard commands can be initiated from Atari joystick controller.

Good Manual and User Support

Optimized Systems Software provides an excellent 25-page manual that explains the theory behind each exercise, outlines sample exercise sessions, and gives simple start-up instructions for first-timers too impatient to read the whole manual. The two disks come with a strongly worded licensing agreement.

OSS has an excellent reputation for support of its Atari operating system and Basic upgrades, and can be expected to stand behind this product. They do promise telephone support, though I was unable to find any significant bugs.

There is a hint also of future updates at reduced rates (or no charge) to licensed users. OSS released the Apple disk version of *SpeedRead+* in May, and they hope to have disk versions available soon for the TRS-80 and IBM PC.

This program is a welcome reminder that home computers can be much more than game machines. I would like to see more personal development tools of this caliber. *SpeedRead+* has much to recommend it, and I do so without hesitation.□

Eastern Front
The Atari Goes to War

Why would a multimillionaire ex-movie star seek a job as President of the United States with a salary of a mere $200,000 a year, or the head of a major corporation join the Cabinet with a salary even lower? The answer is that of all the success drives that captivate the human imagination, the strongest is the lust for power. Power is far headier than sex, wealth, or fame, and may make the others easier to obtain.

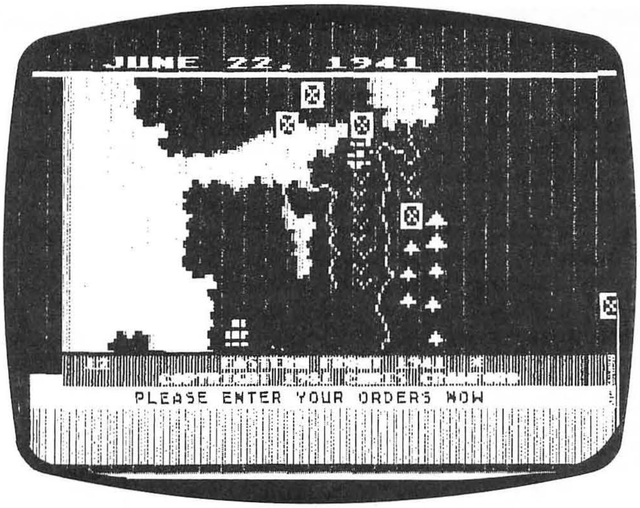
No exercise of power can compare with the job of a commanding general in time of war, marshaling millions of soldiers and the industrial resources of many nations in an all-out drive for supremacy on a battlefield that covers a continent. One of the largest such campaigns in human history was Operation Barbarossa, the German invasion of Russia that began in the summer of 1941. During the course of this four year campaign, nearly 20 million human lives were lost. Eastern Front, one of the best microcomputer war games ever produced, allows the player to take on the role of the commander of the German army, and try to do better than the German forces actually did.

In the past, among war games, board games have had a major advantage over computer games. War gamers like to operate on a theatre level, with an overview of dozens or even hundreds of units scattered over a wide area. Until now the limitations of computer displays have made it difficult to get a satisfying situation map.

Special Features

In Eastern Front, Chris Crawford has produced the first really satisfactory solution to the display problem by using the fast fine-scrolling ability of the Atari computer to produce a magnificent map of Eastern Russia that occupies ten display screens.

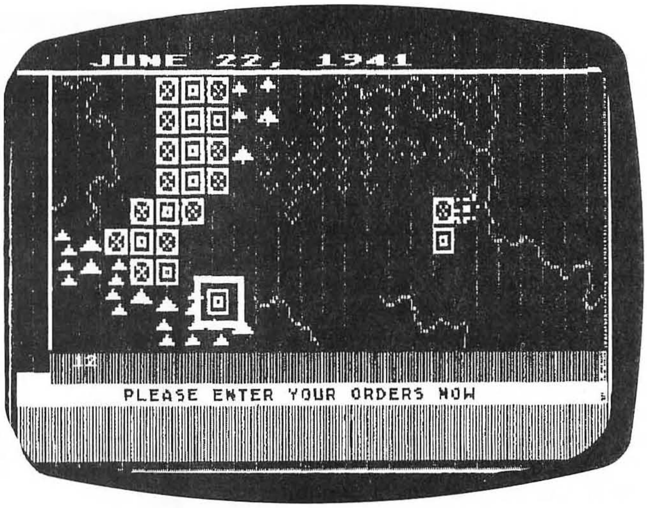
Nearly every aspect of the game is a technical masterpiece. Eighteen colors are



JUNE 22, 1941

PLEASE ENTER YOUR ORDERS NOW

Figure 1. The opening display of Eastern Front shows the Baltic Sea, with two Finnish Infantry Units (German Allies) in Finland and three Russian infantry units. This black and white picture does not distinguish between the units, but the Russians are red and the Axis are white. The city in the top center of the screen, directly below a Russian unit, is Leningrad.



JUNE 22, 1941

PLEASE ENTER YOUR ORDERS NOW

Figure 2. German and Russian units face each other in Central Poland. This display shows mountains, rivers, forests, marshes, and the city of Kiev, along with Russian and German Infantry and Armor units. The cursor is over a Russian Unit. Pressing the button would cause the unit to disappear, identigying the terrain underneath, and also display information on the unit; in this case, the 4th Russian Tank Army, a weak unit with a muster level of 79 and a current strength of 77.

used on the screen at a time. Player missile graphics are used to move a cursor over the map to give instructions without disturbing the map underneath. Several different redefined character sets permit the natural mixing of a colorful and detailed terrain with a text display. Display list interrupts are used to set the weather conditions, with ice gradually taking over the rivers in winter and receding in the spring, and making the player deal with mud and snow at different times in different areas.

The program uses intricate artificial intelligence routines and multiprocessing to control the Russians and their allies. This means that the longer the German player takes to form his strategy, the better the Russian strategy will be. The Russian side can analyze its position, recognize danger and opportunity, avoid traffic jams, recognize the effects of terrain, and plan accordingly.

The human engineering of the game is also a major accomplishment, with all information entered by the player using only the joystick, trigger button, start button, and space bar. This eliminates the drudgery of most war gaming. The multiprocessing even allows the German player to move the cursor around and view different sections of the map while the battles are taking place. Of course, since all battles and movement are real time, it is impossible to see everything that is happening. Excellent sound effects do indicate the extent of the overall action.

The computer adds a great deal to wargaming, particularly by providing a dynamic environment in place of the static nature of board games. Each turn, representing one week of actual time, is broken down into 32 time periods in which units move and fight. Thus a player might program a particular unit to attack an adjacent enemy unit and move toward a city. During the course of a single turn, that unit might destroy the first enemy unit, move forward to engage a unit behind it, force the second enemy to retreat, turn toward the city, and engage in battle a third enemy unit that has come up from the reserves during the turn. Terrain affects both movement and combat, with rivers, forests, marshes, mountians, and cities to complicate strategy.

Playing the Game

At the beginning of the game the German commander has the advantages of concentrated force, short supply lines and superior mobility. However, the Russians have overwhelming numbers, vast territory, and the Russian winter on their side. The object of the game for the German commander is to push as large a force as possible as far East as possible and maintain them. Extra points are awarded for capturing key Russian cities. The Russians are trying to move their forces West, which also affects the German player's score. The score, which is calculated from week to week, can range from 0 to 255 points. It is fairly easy to get a high score by early fall, but nearly impossible to hold that advantage over the winter.

During the war, large concentrations of German troops were bogged down in the Pripet marshes between Minsk and Kiev, allowing the Russians to concentrate their forces. This is a recipe for disaster in the game, as it was also a German disaster in real life. My own best strategies have involved splitting up my forces to prevent the Russians from concentrating theirs, and avoiding combat with superior mobility unless I had overwhelming superiority. Another possibility might be to crash through the Pripet marshes and break into open territory beyond, splitting forces at that time. Uncertain winter supply lines require that the German player draw back during that season.

Regardless of my strategy, my success rate in my first ten games was abysmal. The game ends automatically after the week of March 29, 1942, and in nine of my games my score was 0 on that date. In the one game where I held a score to the end, I seized the city of Leningrad (worth 10 points) and defended it to practically my last man. My total score was 10 points.

After many hours of play, I found only a few real weaknesses. Giving all those instructions with the joystick can give you a sore palm and wrist. The lack of a clear-cut set of victory conditions is frustrating, as is the overwhelming advantage of the Russians. I would also like an option to be able to see the whole theatre at once, however limited the detail might be at that time. The designer mentions in the instructions that test players became frustrated with random logistics problems and traffic jams, but I tend to think these are realistically handled.

Recommendation

I have no hesitation in calling this one of the very best war games available for a personal computer. It is also a virtuoso demonstration of the awesome built-in capabilities of the Atari computer. This game literally could not be done on any other computer in as satisfactory an execution. By all means, if you are at all interested in strategy games, buy it.

If you are a serious war gamer, buy it even if you have to buy a computer in order to run it. Eastern Front comes on disk, requiring 32K of RAM, for $29.95. It is also available on cassette, requiring 16K of RAM, for $26.95. The cassette version can be downloaded from MicroNet at a price of $23.25. □

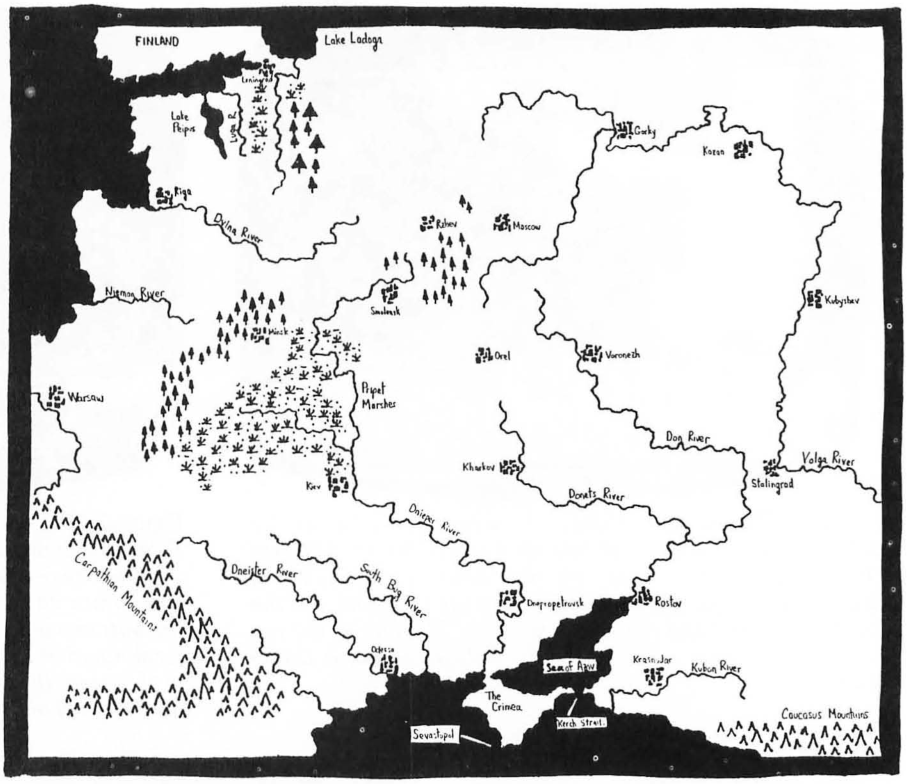


Figure 3. The instruction book contains a map of the whole area covered by the game. Only one tenth of this area is displayed on the screen at one time.

The State of the Art

Missile Command and Asteroids

The Atari personal computer has been around for a couple of years now, and some good software is finally being written for it. For some time, the only software available was (usually) either written in Basic and/or translated from some other machine, usually the Apple. None of these programs really took advantage of the capabilities of the Atari.

Now there are quite a few programs available which use the features of the Atari, not just the subset of them required to translate a program from another machine. They use high speed, quality graphics and sound, and were written specifically for the Atari.

This review will cover two of what we consider "State of the Art" game software for the Atari.

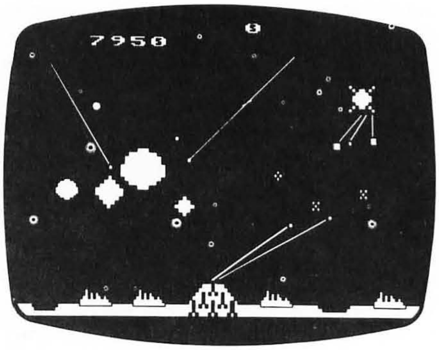
They are from Atari Inc., and are clones of the Atari arcade games Asteroids and Missile Command. Not surprisingly, they bear the same names.

Both are on ROM cartridges which plug into the lefthand slot. Both cost $39.95, and require 16K RAM (no disk needed) and joystick(s).

Missile Command

This is a popular arcade game in which an evil foreign power launches a missile attack against the area you defend. You command anti-ballistic missiles, which you shoot to intercept the incoming missiles, satellites, planes and smart bombs.

In the arcade version, a "trackball" is used to move the cursor for aiming. It allows very high speed movement, and very sensitive positioning. (For example, hitting a "smart missile" exactly on its position is required to destroy it; otherwise the missile dodges). Since no "trackball" exists for the personal computer, a joystick is used.



Missile Command.

Sound effects include an "air raid siren," various explosions, and so forth. They are quite familiar to anyone who has played the arcade game, and make good use of the Atari's capabilities.

Visual effects are also rather well done. There are no longer three missile bases controlled by three buttons, as there are in the arcade version. Instead, there is one, with "underground reloading" which enables it to be destroyed, yet pop up with new missiles a bit later. There are three missile bases in one, all controlled by the joystick button.

The enemy starts with single missiles, moving slowly, then escalates to MIRV's (missiles which break into multiple missiles), satellites and planes (both of which drop missiles), and finally smart bombs which dodge explosions on the way down. Everything begins to move faster, the bombs get more dense, and so forth, until you are finally overwhelmed. As in the arcade version a bonus city is awarded for every 10,000 points.

There are several variations of missile command. An attack consisting solely of smart bombs can be ordered up, if desired, to allow practice with them (a very useful option). There is also a two-player version, and an option to "freeze" the game if you want to get another beer.

Rating

I rated this game the better of the two. It is excellently done with one exception, and that's the joystick handler. I found it very difficult to position the cursor precisely.

The problem is twofold. First, the cursor moves up/down/right/left at the same speed, but moves diagonally as a double increment of up-right, down-left, etc. This makes the diagonal move functionally faster than the others, which makes linear motion darn near impossible. I found myself firing multiple missiles near the same point, and constantly missing. The fine control of the arcade version was missing.

I'm not sure how this could be changed. Perhaps the diagonals could be slowed down a bit and some sort of fine position enabled, with coarse movement occurring a bit later on the same joystick press.

I found the home game just as challenging as the arcade version; my top score seems to be limited by not being able to position the cursor with enough accuracy. (Particularly important with smart bombs.)

Despite my reservations, this is a good game. It's not a replacement for Star Raiders, but it is well done and fun to play. Nor does it get boring after a few turns. I recommend it.

Asteroids

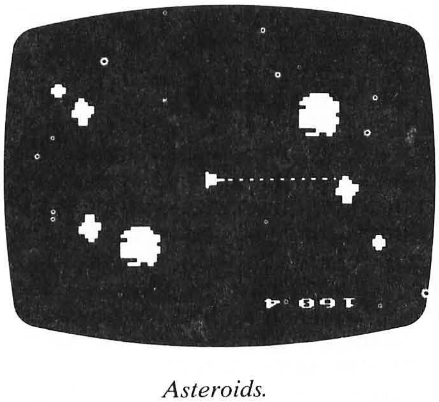
As an addict of the arcade version of Asteroids, I really looked forward to this game. I had begun to design an Asteroids game for the Atari (laid out the player shapes and so forth, and had the basic algorithms worked out), but when I heard Atari was releasing a version, I gave up.

I'm not sure I should have.

Asteroids, as you probably know, is a game which places you in a ship in an asteroid field. You shoot at the asteroids, which break into smaller asteroids, and try to avoid collisions. Occasionally an enemy ship enters the field and fires at you.

This version of Asteroids is apparently written in graphics mode 7 (Basic) or Antic mode 13. This means it has a "chunky" feeling to its graphics. If you have ever played TRS-80 asteroids you know what I'm talking about.

This is particularly surprising when mode 14 is available (graphics 7 1/2) with much better four-color resolution. Indeed, I had planned to use this mode for my version and include three different colors of asteroids. Even graphics 8 (Antic 15) would be a possibility if multicolor asteroids were not required.



Asteroids.

Anyway, I find the low resolution look of the asteroids quite annoying. Also irritating is the very large distance between "turn points" on the ship; in other words, a minimum turn is a large distance.

The missiles are limited to four and probably not done with P-M graphics, as there is an option for up to four players at once. Ah, well.

The joystick is used as follows: right and left are rotate, forward is thrust, back is hyper/flip, your 180 degrees/shields. The shields are not "timed" as in Deluxe Asteroids, by the way, making for a rather predictable game.

Rating

Alas, this one is not as good as Missile Command. I liked it, but not enough, and it could have been done better. Possibly the video game version and this version were made as similar as possible to cut development costs. I can understand the problems, having worked this out myself (for example, how to rotate a rocket in only 8 bits; it looks pretty weird in some angles), but still, much better resolution could have been achieved.

The multi-player option is a lot of fun, and my wife and I spend much time shooting at each other.

One thing you will notice, again, on most Atari games is that they are not CPU bound. On a version written for another machine, there is a very noticeable slowing of the game when there are many asteroids present. This is the result of all the table updating, checking for collisions, and so forth. The Atari version runs at a constant speed, and is fast.

Summary: I play Missile Command much more than Asteroids.

Conclusion

All in all, these were fun games to play. Asteroids will entertain those of you not spoiled by the arcade version, which I admittedly am. It is a good sign that these games exist, as it means that more good software for the Atari is becoming available. □

Star Raiders and the Atari SOUND Command

Star Raiders

That is the excuse you give the Internal Revenue Service, your accountant, and your husband. Truthfully, the reason you bought your Atari was to play Star Raiders (TM), the most addictive computer game yet developed.

The game comes as a ROM cartridge with a lavishly illustrated twelve page instruction manual at a cost of $59.95. In addition, you need to purchase a joystick, costing $19.95 for two. The joysticks are not sturdy, and get heavy use, so you can use the spare.

Study of the instruction manual takes about 45 minutes and is essential to adequately understand the game. However, if you have someone available who already knows Star Raiders, it can be learned in 5 minutes by demonstration, if the demonstrator will then give up the machine. That brings you to the point of understanding. To truly master the game might take years.

Your mission is to defend your star bases from the Zylon fighters. You do this by locating the enemy on the Galactic Chart, turning on your defense shields, hyperwarping through space to the enemies' sector, and engaging them in combat until the best man, woman, or Zylon wins.

You are rated upon your performance based upon the level of play you have chosen, the number of enemy destroyed, the length of time it took you, the number of your starbases that have been destroyed, and the amount of energy you used. Final ratings range from Garbage Scow Captain, class five to Star Commander, class one, with 60 different possible ratings. There are four levels of play, from novice to commander.

The graphics and the sound effects are brilliant. Stars whiz past you, your engines whoosh and your torpedoes explode, your klaxon sounds a red alert, and the enemy fighters speed past you, coming from all angles and all sides, firing their exploding torpedoes. Enemy fighters explode in clouds of blue particles, while the sky flashes red whenever you sustain a hit.

The instrumentation of your ship is also impressive. In addition to your Galactic Chart, which is updated by sub-space radio, your color coded instruments tell you the range to the enemy being tracked on the x, y, and z axis, your velocity, shield status, energy level, the condition of your photon torpedoes, engines, computer, long range scan, and your sub space radio. Your target aquisition computer helps you to steer while hyperwarping through space, as well as indicating the relative position and range to enemy fighters. In addition, upon your request it will shift automatically from forward to aft views from your ship as enemy fighters pass by on attack runs. The joystick allows you to climb, dive, veer right and left, and to combine vertical and horizontal movement, while twenty more keys on the keyboard control speed and function selection.

Star Raiders requires a color monitor or television, as much of the information is color coded and does not show up in black and white. I cannot pin down any definite bugs, although it is often hard to orbit a star base, and I did have a system lockup once in the middle of a game that required me to turn the power off and on again and restart the game.

This game goes beyond the quality of the games you see in video arcades. The sound effects, color, and action are just as good, the physical environment is a bit less impressive, but the real change is the strategy. Since an arcade game must produce $10 an hour in revenue, those games have to be active and short. Grand strategy is not possible. A home computer does not suffer from the same constraint, so the game can actually be better, and Star Raiders is better. The true video arcade addict can justify the purchase of an Atari 400 in a few months of unspent quarters.

If you have an Atari, buy this game! If you don't have an Atari, sell your car (you'll never leave home again anyway), put your children up for adoption so they won't take over the computer, and buy one. Then play Star Raiders until the last stardate fades into the collapse of the universe.

Atari Sound

As a programming feature this month, I'd like to discuss the Atari SOUND command. The format for the sound command is as follows:

SOUND (Voice) , (Pitch) , (Distortion) , (Volume)

You can have up to four voices, or notes, that can be played at the same time, numbered from 0 to 3. Each voice is totally independent of the others.

Pitch can range from 0 to 255, with high C at 29 and low C at 243. Distortion (timbre) can take any even number from 0 to 14. The value 10 gives a pure tone, while other values are used for sound effects. Volume can range from 1, which is hard to hear, to a loud 15. If you are using three or four voices, you should limit the total volume to 32 or less to avoid distortion. To turn the sound off, use the command END or set the volume for that voice to 0.

Photo 1

This program will demonstrate the range of sound available, displaying the value on the screen so that you can note sound effects you would like to use. Really good sound effects will mix several voices.

```
10 FOR A = 0 TO 14 STEP 2
20  FOR B = 0 TO 255
30   SOUND 0, B, A, 8
40   PRINT"SOUND 0, ";B;", ";A;", 8"
50    FOR C = 1 TO 250 : NEXT C
60  NEXT B
70 NEXT A
```

I like SOUND 0, 6, 0, 8 : SOUND 1, 21, 0, 8 : SOUND 2, 27, 0, 8 : SOUND 3, 40, 0, 8 for an explosion, SOUND 0, 17, 8, 8 for a Phaser, SOUND 0, 30, 8, 14 for a gun shot, SOUND 0, 70, 2, 8 for a truck motor, SOUND 0, 145, 2, (1 to 12 to 1) for an airplane motor, and SOUND 0, 12, 4, 10 for a machine gun, but I am sure you will have your own choices.

Basketball

One of the first Atari games is still one of the best. In Atari Basketball, you use the joystock controller to move around the court, dribble, shoot, pass, block shots, and steal the ball. The exceptional graphics and animation of this game make it a favorite demonstrator at computer stores, so many of you have already seen it.

How well does it play? The answer is that it is relatively easy to beat, but not easy to trounce. The computer is set up to play better when it is behind than it does when it is ahead, so it offers a good challenge until you get really good. However, once you can consistently trounce the computer, you've only begun the real fun!

The best feature of Basketball is that it allows one to four people to play at the same time. There are five options:

1. One player against the computer
2. Two players against the computer
3. Two players against one player and the computer
4. Two players against two players (no computer player)
5. One player against one player (no computer player).

After all, if you let your best friend play Star Raiders, it may be weeks before you get a chance at the computer again! With Basketball, you can both play at the same time. Teams of two are even more fun. This is one of the best computer games available for more than one player.

Basketball requires one joystick controller for each person playing and is available for $39.95.

Warlock's Revenge and Kayos
Dungeons and Asteroids

Warlock's Revenge

Warlock's Revenge is an Atari translation of an Apple game, *Oldorf's Revenge*. It is another graphics adventure and seems well done. I didn't encounter any bugs in my playing of it, and I had a good time, although I have to admit I'm beginning to burn out on generic adventure games.

After a certain point, you see, I get tired of trying to figure out which implement I must use to get past a certain point. The game becomes boring, and settles into mere combination testing. While *Warlock's Revenge* suffers from this malady to some extent, it isn't nearly as bad as some I have seen. It wins points for this; there's nothing worse than an unplayable, un-figure-outable adventure.

In this game, you are leading a party into a dungeon. You can be any of several different types of character (cleric, magician, and so on), each of which has special skills. These skills are needed to get past a certain point in the dungeon and to continue the adventure. Be prepared for a great deal of testing of combinations, or perhaps a short session of dumping the game database to the printer. Hint: the game is all *hardcoded*, with all pictures, etc., coded into the program.

The pictures are all done in graphics 8, the highest resolution mode the Atari has. They seem to have had a good amount of work put into them, and the only detraction is that in graphics 8 the Atari doesn't put out a solid line, it tends to candy-stripe and change colors. This is called artifacting and can be of use to a programmer who understands it; the folks who did *Warlock* didn't, I'm afraid, so you would do well to turn off the color on your TV.

The game itself is a fairly standard adventure, with pictures at each stop and two-word commands. It runs fast enough and is fun to play. I recommend it and had a good time playing it, even if (I must confess) I have yet to completely finish it. This one will take you more than a couple hours to do.

In summary, while it may be "just another adventure," the game is a lot of fun and good to play. Don't let the fact that there is good competition for it worry you; just because there are several good games like it available, doesn't mean this one isn't worth getting. There aren't yet enough adventures on the market to swamp it completely, so if you're into such things, or if you would just like to give one a try, this is a good choice.

SOFTWARE PROFILE

Name: Warlock's Revenge

Type: Adventure with Hi-Res screens

System: Atari 400/800

Format: Cassette or disk

Language: Basic and machine

Summary: A good adventure game

Price: $34.95

Manufacturer:
Synergistic Software
5221 120th Ave. SE
Bellevue, WA 98006

Every reviewer has to fight a tendency to be sarcastic when he discovers a game that just doesn't make it. The urge to make cutting comments can be overpowering. In this case I was going to award the Cray-1 Speed In Arcade Games Trophy for this game. But that isn't how I view the purpose of a review. I prefer to try to make constructive comments on games that aren't quite right in the hope that the author(s) will consider my opinions and suggestions and, perhaps, improve the game.

Kayos

So we come to *Kayos*. You've guessed it—it doesn't make it. It is very well done technically. It runs faster than most, and obviously a great deal of work went into it. I have no complaint with it technically. However, its human interface isn't very good. It is simply too fast for people.

SOFTWARE PROFILE

Name: Kayos

Type: Arcade

System: Atari 400/800

Format: Cassette or disk

Language: Machine

Summary: Good if you have
superhuman reflexes

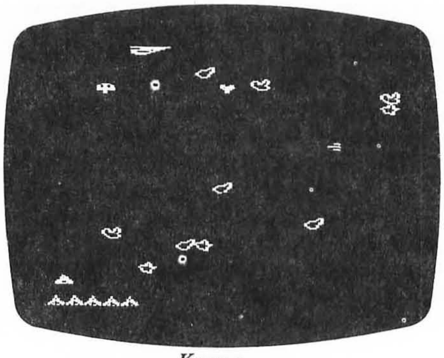
Price: $34.95

Manufacturer:
Computer Magic Ltd.
P.O. Box 2634
Huntington Station, NY 11746

Robots with emitter-coupled-logic reflexes might enjoy it, however.

When you boot it up, you see a field of asteroids crossing space from left to right—a complex animation task for sure; someone worked very hard on it.

At blinding speed a series of blurry objects comes out of the top of the screen and dives upon your emplacement; I could never identify what they were, they went so fast. My average playing time was around a minute or two, and I just couldn't see spending too much time on the game.



Kayos.

Galaxians, and arcade games like it, are a challenge because they are not too fast. Much fiendish design effort went into making them just fast enough to be an agonizing challenge and not simply impossible. *Kayos* lacks this human engineering quality. It is a game sadly in need of a few strategically placed delay loops.

Look for a reissue soon, I hope. This could be a fun game if it were slowed to a playable speed. □

Gamma Hockey
Getting Iced

Norman Schreiber and Witold Urbanowitz

SOFTWARE PROFILE

Name: Hockey
Type: Two player sports action game
System: Atari 400/800 16K
Format: Disk
Language: Machine
Summary: Achieves its goal
Price: $29.95
Manufacturer:
Gamma Software
P.O. Box 25625
Los Angeles, CA 90025

Hockey, by Gamma Software, is almost fast, never furious, and generally fun. That's what we—Witold, Norman, Roman and Jason—discovered one Stanley Cup weekend.

We loaded the game (which, by the way, requires 16K), powered up, and the screen beckoned with a menu of options—nine in all. (Game durations are three, five or eight minutes and two, three or four people can funnel their hostile energy into knocking a puck across the ice.)

Each game begins with the last tones of the "Star Spangled Banner," followed immediately by the roar of the crowd. It's four on four as one goalie and three free-skating forwards go against each other.

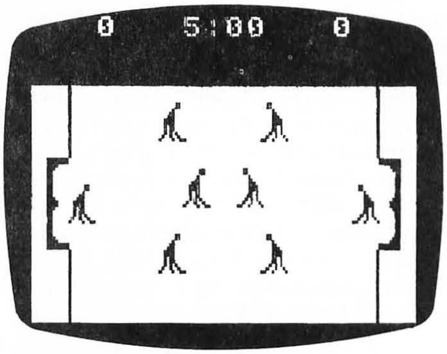
Using joysticks the human opponents manage the teams. With the three-player option two (one controlling the goalie) gang up on the third. Four-player play brings both goalies under joystick control. A scoreboard and clock sit at the top of the screen.

We first selected a two-player, three-minute game. The puck was dropped and the two center forwards, under joystick control, went into action. Each goalie's movement in front of the net corresponded with up and down movements of the joystick. The other four forwards moved as "smart" players.

The action was intense. The hockey puck slid and caromed across the horizontal ice, and the players scrambled to dominate the puck. Joystick control remained with the original two until the puck struck another player's stick. And *voila*, the joystick managed that player. When the puck was free, the joysticks controlled the original center forwards. This created opportunities for some fancy passing, a neat way to outsmart the opponent or even oneself.

Inevitably, the action brought the players, in one Gamma glut, directly in front of a net. A shot was taken. The goalie edged sidewards and successfully blocked. Another head-on shot brought another block. The next try started from the corner and homed in at a sharp angle. It whizzed past the goalie and the crowd roared. Players reassembled at center-ice for a new face-off. And so on.

At game's end, the score was tied, so we were thrown into sudden death overtime and given an additional three minutes. Unfortunately neither glorious team could score. There was no additional overtime, so we settled for a tie and celebrated with a rematch.



Gamma *Hockey* arouses competition. The four of us scarcely kept our tails upon our seats as we played the game at various angles of leanforwardness. The value of the three- and four-player options was that we adults could also get into the game, rather than just hover. Actually, the four-hand participation did make the game that much more exciting. It is unfortunate, however, that no solitaire

Norman Schreiber and Witold Urbanowitz, 135 Eastern Parkway, Brooklyn, NY 11238.

option exists. One would like to get one's chops together in the quiet of one's own fantasies.

The two teams are blue and green; except on a black-and-white monitor in which case they are grey and grey. You can tell who is on first by the direction in which the hockey stick points. The thoughtful designer(s) made joystick-controlled players flash when the puck was free. However, the "smart" players tend to flicker as they move. Consequently, there was a certain amount of confusion at certain points.

Perhaps the most intriguing, at least to reviewers, aspect of Gamma *Hockey* is the slow rate at which the players move speedily. Said nine-year-old Roman, "Can't you make them go faster?"

Said 35-year-old Witold, "It might have something to do with the horizontal movement of the game in what is basically a vertically-structured medium."

Norman, ·41-years-old, observed that there was no way for players on one team to maim, destroy or righteously punish players on the other team. (This happens to be his favorite feature in the Activision VCS cartridge.) And 14-year-old Jason kept on scoring goals.

Something should be said for and against the sound. It keeps the game going and provides some pleasant texture for the ongoing battle. However, after playing 12 games, one gets to feel a bit unpatriotic at wishing the familiar notes of the national anthem would speed up radically (pardon the expression). Perhaps if there were a Kate Smith voice chip things would be different. Perhaps not.

The crowd noise, though useful, sounds suspiciously like our television sets at four in the morning when there's nothing to pick up but noise, and made us wish for a Dolby override.

We also should note that during one of our many games, four players suddenly disappeared. They could not be found. They certainly weren't in the penalty box. They just as mysteriously reappeared in a few seconds. We tried to render the hockey players invisible again; and failed. We haven't the foggiest notion why this happened. Not even Witold has a theory. Final note: The documentation is clear, concise, easy-to-read, and offers some useful tips.

Postscript: We had to go through all the options. After all, we decided, we really had to explore the game. We owed that much to our readers. We would have ended the tests sooner, but regardless of which time option we played, the final buzzer always went off too soon. □

The Wizard, the Princess, and the Atari

A copy of *The Wizard and the Princess* for the Atari recently appeared in my mailbox. This happened around Christmas time, and the family was visiting. So I decided to show them the game, and soon the whole Small clan became involved in playing and trying to beat it.

It took us roughly four solid days to do so.

The Wizard and the Princess comes attractively packaged with the disk adequately protected against any but the worst of Post Office Bend-a-Disk equipment. The directions are on the printed folder surrounding the disk and are clear enough. So, you boot the disk without a cartridge, for the game is written entirely in 6502 machine code.

This is its first plus mark. One of the least endearing features of some programs is Atari Basic, one of the slowest executing languages ever developed. *The Wizard and the Princess* runs very quickly and with minimal delay.

SOFTWARE PROFILE

Name: The Wizard and the Princess

Type: Adventure game with pictures

System: Atari 800 version
Requires 40K RAM memory

Format: Disk, double sided

Summary: Very good adventure game with graphics

Price: $29.95

Manufacturer:
On-Line Systems
36575 Mudge Rd.
Coarsegold, CA 93614

Next, you boot the system, and wait for the driver routine to load and for the disk protection scheme to determine that you haven't copied the disk from someone else. Then, you are told to flip the disk and insert the reverse side.

Backup

The data tables, hi-res screens, and all are on the flip side of the disk. The flip side isn't copy protected, and given the amount of time the disk head spends beating on it, it has a good chance of failing, so it should be backed up.

The folks at On-Line have thought of this, and they provide a backup routine. If you boot up off of the back side, you are automatically taken to a backup routine, which will format a new disk and copy itself—very nicely done, very convenient, and very thoughtful.

The disk spends most of its time on the flip side, and is in almost constant use while the game is being played. This is the only slow feature of the game. Atari disks are 1/20th as fast as Apple disks (serial vs. parallel) and it shows, even though attempts have obviously been made to minimize the problem. For example, the *W&P* disk is fast-formatted to allow faster disk access.

Snakes Alive

As we began the game, we wandered out of Selenia northwards in pursuit of the wizard, and immediately ran into a rattlesnake which wouldn't let us by.

Being an old adventurer, I knew I needed something to get by him, but nothing I had on me worked, so I set off south in search of the proper object. Aha, a rock.

I picked up the rock, and died, for the first of many times, after being bitten by the scorpion hiding behind it.

Many hours later the family figured out how to get by the rattlesnake; it is one of the most difficult parts of the game. Fortunately, the authors had included a hint card, labeled "How To Get By The Rattlesnake," which helped considerably. (Naturally, we didn't read it until we were so frustrated we were ready to burn the disk.)

With the aid of a good deal of mapping, we proceeded on our way, picking up everything imaginable.

A hint to players of this game is to LOOK at everything you pick up; some of the most subtle hints are there. We hate to give away any of them, but do be sure to LOOK at everything; we wouldn't have gotten stuck in a few places if we had done so.

On the way north, we had to cross a bridge, fend off another snake, outsneak a gnome, figure out several magic words, learn how to operate a rowboat (and how to plug the hole in it), find an island, and do many other wonderful things.

In terms of difficulty I would rate *The Wizard and the Princess* right up there with some of Scott Adams's efforts, and the high-res screens add a new dimension that is a great deal of fun (even when everything is green and blue). As I said, it took our family four days, and that's only because there were many people adding new ideas all the time; one person might need weeks to finish this adventure.

Finally, after much mapmaking, meeting of old peasant women, buying peddlers' wares, and dying, we made it to the castle, confronted the wizard, and rescued the princess, bringing her safely back to Selenia.

Chameleon Chips: CTIA and GTIA

I was very surprised when the hi-res screens of *The Wizard and The Princess* turned out to consist of shades of four-hours-of-turbulence-in-a-DC-10 Green, Mental Hospital Blue, and you've-just-crashed-the-Atari black. Those were the only colors, although some of them were shaded by interspersing dots with other colors. This was very disappointing. How could it be?

This color business annoyed me a great deal. I have put 128 colors of all sorts of neat red, green, blue, and orange shades onscreen, making full use of the abilities of the Atari, and the authors appeared not even to have tried to take advantage of these same abilities.

The reasons behind the displayed colors are rather complex and worth pursuing, for other games will suffer from the same malady: it is the effect of a new graphics chip on what is known as "artifacting."

In graphics 8, a single dot by itself will appear to be either blue, red, or white, depending on where it is written and its proximity to other dots. (And here you thought graphics 8 was a single color mode, like the book said).

This happens because of "artifacting," which I understand to be the Atari running the TV out of resolution and ending up with a color other than what is normally output. With careful use of graphics 8, one can get four colors in this hi-res mode.

The authors of *The Wizard and the Princess* originally wrote it for the Apple, which has a 280 x 192 screen. So the tables for the hi-res drawings were scaled appropriately. The two highest-resolution modes of the Atari are graphics 7 plus, with 160 x 192, and graphics 8, with 320 x 192. In graphics 7 plus, various colors can be plugged into the color registers; in graphics 8 the colors result from artifacting.

The person who translated this program from the Apple to the Atari had the choice of scaling down all the tables to 160 x 192 and using graphics 7 plus, or using graphics 8 with artifacting and the table data unchanged. He chose the second approach, so the colors are the result of artifacting.

Other manufacturers use this approach, as well. For example, *Jawbreakers* uses artifacting to color the playing field. The problem comes when artifacting is used with the GTIA chip, a new graphics chip recently released by Atari. The GTIA chip replaces the CTIA chip; both are the essential color television driver circuits for the Atari.

The GTIA has more graphics modes, and in my experience, gives a sharper display, than the CTIA. As of January 1981, Atari has shipped all Atari 800s with the GTIA chip.

There is just one drawback: the GTIA artifacts differently from the CTIA chip, and *The Wizard and the Princess* was written for the CTIA chip. Sandy and I had a GTIA chip in our Atari, so instead of the colors the author used, we got the particularly grim shades of green and blue.

I called On-Line and mentioned the problem. A fix was already in the works for GTIA machines. In the new version (which I haven't seen) a box is drawn onscreen and the player is asked if it is green or orange. Depending on which chip is installed, it will be one of those colors. The program then generates the correct colors from the color tables based on that information.

I mention all this about the GTIA chip because there may still be some of the older *Wizard and the Princess* disks being sold. If you have a new Atari, you will get a GTIA chip and the same terrible colors with an old version of the game. However, the people at On-Line are friendly and willing to swap disks if you have a GTIA chip, and don't want to see green.

Atari is making the GTIA chip available to CTIA machine owners. The upgrade will be performed at Atari service centers, or you may choose to buy the chip outright to do the installation yourself. Atari owners choosing this option should be aware that removing the bottom cover may void all warranties. To obtain the address and phone number of your nearest Atari service center, you can call Atari at (800)538-8547 (outside California) or (800)672-1430 (in California).—*DS*

We are still waiting for our half of the King's land, though.

Features

The hi-res pictures are good, often with good detail on them; the ideas are original, and require some thinking, which is also good; and the implementation is generally good, even if it is bit disk-dependent. There is something to be said for the idea that text-only games force you to use your imagination more than the versions with pictures; we enjoy them both.

There is a very nice "save game" feature which allows you to save the game at any point. You put in a blank disk, and type a letter, A-L, which labels the saved version. At any point thereafter, you can RESTORE GAME to any of the saved versions.

The ability to save multiple versions on one disk is very nice (Scott Adams take note) and we used it a great deal. You can even initialize a new disk from inside the game—a very professional touch.

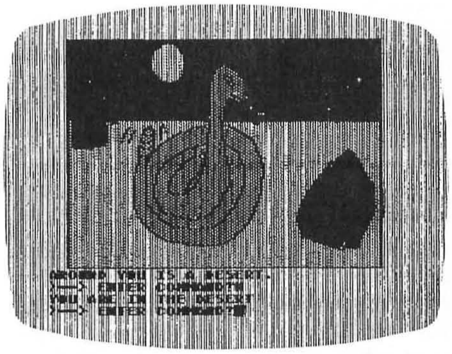
One Big Complaint

Now for the complaints. I have one major complaint about the game, and it is a very subtle annoyance. Like most minor irritants, however, it builds up over time until it gets to the point where you can't stand it anymore.

Unlike a really gross deficiency in the game, such as an execution error or problem of that sort, this one takes a while to get on your nerves, but its effect is devastating.

The game has a four-line window at the bottom of the screen, in which all displayed text is shown. Often the text won't fit into four lines, so the authors have the machine pause in the middle and wait for a Return keypress. After that keypress, the output continues. If you press any other key but Return, you get a beep. This is a particularly awful sound, which makes you suspect that the POKEY sound chip is being flogged.

The beep is to let you know that what you are typing—typically the next command—isn't being accepted by the machine. It lets you know that the machine wants a Return before you can go on. No other key will satisfy it.



```
AROUND YOU IS A DESERT.
---> ENTER COMMAND??
YOU ARE IN THE DESERT
---> ENTER COMMAND?
```

It is particularly irritating when the ouput from the machine finishes midway through the four-line text window. You assume it has said whatever it had to say ("The peasant woman warns you of a giant in the mountains"). So you would begin to enter a new line, and are rewarded with this awful beeee—eee—eeep.

You must then patiently hit Return and start all over.

"Frustrating" isn't the word. "Annoying" isn't, either. After the eightieth time it happened, I gave up and stomped off. My sister Diane, who is the epitome of patience and calmness, took over. She lasted until the hundredth beep at which point we had to restrain her from throwing the Atari into the TV.

Our nerves grew jangled. Our parents left for a nice, long, soothing walk away from the noise. Sandy and I started snarling unprintable things at each other. The dog began to howl after each beep.

Finally, I couldn't handle it anymore. So I went to my tool kit, almost picked up a hammer, but decided that there was a better way. I got a screwdriver, removed the bottom cover, and disconnected the speaker.

Once the speaker was disconnected (Remove the five lower screws on the Atari, pull the speaker plug off the jack, and reassemble), things improved. The whole mood of the family changed. Our parents returned. I gradually regained my sanity. Diane became calm, cool and collected once again. The dog even shut up. And we realized just how much the sound had annoyed us.

After this change, we settled down and really got to work. We enjoyed it immensely. The family's computer experience rated from very high to none, and all enjoyed the game equally. (In fact, those with the least experience often supplied the ideas to get around obstacles.) And after four days, we finally won.

We recommend it to Atari owners who want to try their hands at a little classic adventuring. We also recommend a phillips head screwdriver, to disconnect the speaker, if the beep feature hasn't been changed.

But all in all, it was a lot of fun to play, and well worth the price. Of the adventures available today, it is unique and very interesting—a real challenge. □

Graphics Adventures on the Atari

John Anderson

Adventure games, an established and popular genre among microcomputer enthusiasts, are generally divided into two categories: text adventures and graphics adventures. This dichotomy seems clear enough: a text adventure uses words alone, similar to books or radio plays, to create a picture in the mind. A graphic adventure, in contrast, draws these pictures for the eye. Each category claims its own adherents.

Text adventure aficionados assert that only verbal descriptions can provide a satisfactorily rich level of story telling, as they leave most of the visualization to imagination. For these people, a good adventure is like a good book—a reading experience.

Graphic adventure chauvinists point out correctly that text adventures employ a very static screen display. This is wasteful, they argue, in light of the capabilities of microcomputers. Graphics potential (and sound potential, for that matter), should not be left unexploited by the adventure program. Each faction has a well-taken point.

The fact is, however, that the division between text adventures and graphic adventures is not nearly as sharp as this, and some of the most interesting developments in microgames are taking place between the two poles. All graphics

John Anderson is an associate editor for *Creative Computing* magazine.

adventures employ text to one degree or another—either to augment the graphic display or as the basis for graphic augmentation. Let's examine these approaches more closely.

One method of constructing graphic adventures might be called the *illustrated text adventure*. This employs all the normal conventions of a text adventure, including text command input, though the text may be less descriptive and more to the point. In addition to this, the player is provided with an illustration of his or her current position via a high resolution picture. These are stored on disk and are called out as necessary by the main, text-oriented program.

The simile is close to the idea of a comic book. Each piece of the story has its own "frame" of picture and text. The point of view is that of the central character—the player sees the locations as if he were there.

Another approach results in the *mapped adventure*, wherein the player or players appear as symbols on a map. This map depicts the details of the location, indicating the type and placement of terrain, walls, objects, and enemies, among other things. Player movement is input through keyboard or joystick. As a character reaches the border of a screen map in any direction, either the screen begins to scroll in that direction, or a new location map is drawn. This creates a more omniscient perspective, with the player looking down on his character's movement from above.

Typically this type of adventure does not allow text input, but limits the player to a certain number of possible contingencies in a menu or command format. The comparison between text adventures and mapped adventures is close to that of a fill-in-the-blank vs. a multiple-choice test.

Text adventure enthusiasts feel the menu format is restrictive—one can only choose to move or stand still, flee or fight, take or drop, and so on. The mapped adventure began as an outgrowth of a certain fantasy role-playing game whose name we are not permitted to print. One obvious advantage of this format is the capability for multicharacter play.

Until quite recently, the preference boiled down to a choice between the classic-style text adventure game and the fantasy role-playing adventure. However, hybrids are now being developed that now provide both types of enjoyment.

Truly hybrid adventure games of the future will offer the best aspects of all approaches to computer gaming. Textual description will provide detailed background and help set the mood. Maps will be available to indicate position (except in areas still unexplored). Computer graphics and sound will be called on where appropriate to animate action sequences. Arcade-style challenges will become part of the stories, calling for feats of coordination before the plot advances. And it won't be too long before the videodisc becomes a necessary peripheral for state-of-the-art adventuring.

This is still a ways off in the future. That is quite enough background, however, to examine the spectrum of graphic adventure software currently available for the Atari.

Mission: Asteroid

In the arcade game *Asteroids*, you use hand-eye coordination to zap as many asteroids as possible in the few minutes you are allotted. In *Mission: Asteroid*, your job is to destroy just one asteroid, and you have several hours in which to do it. The game allows you to input two-word commands, and displays a high resolution picture for every location, with a four-line text window at the bottom of the screen. Pressing return without entering text will kill the picture momentarily, giving you a chance to recall the last 24 lines of the text that have scrolled by. Tap return again and the picture will reappear.

The plot line of *Mission: Asteroid* is pretty easy to follow; it was designed as an introduction to a series of "Hi-Res Adventures" of much greater complexity. That is not to say that the game is easy to solve—it's not.

Briefly described, you are an astronaut, who is sent into deep space to destroy an asteroid headed toward earth. You have only a limited amount of time before the asteroid collides with earth and the game is over.

SOFTWARE PROFILE

Name: Mission: Asteroid (Hi-Res Adventure #0)

Type: Space Adventure

System: 40K Atari 400/800

Format: Disk

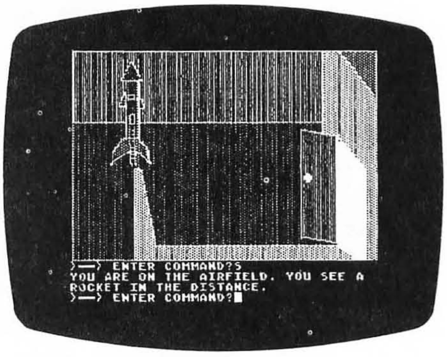
Language: Machine

Summary: Introduction to a series of graphic adventures

Price: $24.95

Manufacturer:
On-Line Systems
36574 Mudge Ranch Road
Coarsegold, CA 93614

It is interesting to gauge the reactions of text adventure buffs to the over 100 pictures on the disk. Some enjoy them thoroughly, others use them as mnemonic devices, which eliminate the necessity to map the adventure on paper. Still others find the pictures a questionable, if not downright distracting, addition. The pictures are well-done—obviously a lot of time was spent executing them. They are pretty, but they do not achieve even the quality of a well-drawn comic book. This leaves me a little dissatisfied. I think it is fair to ask of any graphic adventure game: How well would the adventure stand on its own, if we were to delete all the pictures?



```
>—> ENTER COMMAND?S
YOU ARE ON THE AIRFIELD. YOU SEE A
ROCKET IN THE DISTANCE.
>—> ENTER COMMAND?█
```

Mission: Asteroid.

In the case of *Mission: Asteroid*, the plot line is kept intentionally simple, as it is intended to be an introduction to a series of graphic adventures, and it is an enjoyable program. As such, it would not be really entertaining without the pictures. The graphics serve to enhance the overall effect, but are unable to enrich the story significantly. In the intricate and involving stories of other adventures in the series, they stand a better chance of achieving this goal. I am certainly among those who feel illustrations can improve text adventures.

Apparently so is Scott Adams, who is in the process of re-releasing all twelve of his now-classic adventures as illustrated text adventures.

Ali Baba

Ali Baba and the Forty Thieves attempts to move beyond the illustrated text adventure. It was designed for the Atari, in contrast to *Mission: Asteroid*, which is an Apple translation. The game makes use of some of the special features of the Atari, such as multi-channel sound. It is an example of a mapped adventure, wherein players are depicted on a multicolored map. As the players move to new

SOFTWARE PROFILE

Name: Ali Baba and the Forty Thieves

Type: Mapped-style graphic adventure

System: 32K Atari 400/800

Format: Disk

Language: Machine

Summary: A fantasy role-playing adventure with some new ideas

Price: $32.95

Manufacturer:
Quality Software
6660 Reseda, Suite 105
Reseda, CA 91335

locations, new areas of the map are displayed.

When you first sit down with *Ali Baba*, you may not get up for several hours. The game shows strong potential when you see it for the first time. It allows for a total of 17 characters to participate simultaneously in a search to rescue a kidnapped princess.

This adventure leans heavily in the direction of a fantasy role-playing game, assigning weights to the attributes of each character.

From my experience, this makes for a much more lively game when friends are sitting in. Instead of everyone in the room discussing what the sole character in a text adventure should do, everyone can be his own character, interacting within the adventure as well as with each other. The discussion takes on new depth as individuals decide what course to take for themselves.

In *Ali Baba* you can choose to become one of the set of humans, elves, halflings or dwarves that are profiled in the extensive documentation that accompanies the game. Each of these sets of beings is represented by a different symbol on the screen. Using the keyboard or joysticks to input movement, players can explore the caverns, palaces, passages, and treasure rooms of the game—and happen upon the dangerous inhabitants therein.

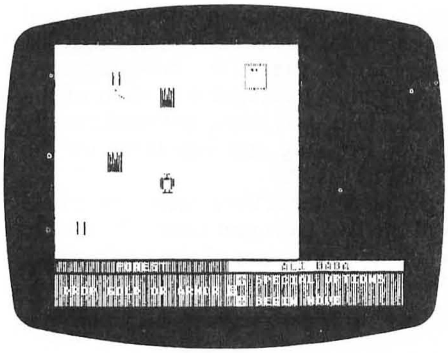
The game is immediately addictive, and is really tough. The first thing the novice should do is turn the "monster recurrence" level down to zero. This will keep him from becoming utterly bogged down in fending off attackers.

You'll spend hours wandering around, acquiring treasures, buying weapons and armor, fighting enemies and thieves, and searching for the princess. Characters can be reincarnated if they are snuffed out, and reinforcements can be called in when the going gets really heavy.

As in other role-playing games, each character has unique attributes. Elves are fast and hard to hit; dwarves are slow and clumsy but when they connect they pack a heavy blow. This richness of character is the strong point of the game.

The adventure has some other interesting features. It can be configured to play through nested menus, controlled completely by one or more joysticks, eliminating keyboard entry completely. Characters can be retired as well as reincarnated, and in fact Ali himself can be retired if no one wishes to take his role. Characters other than players wander through the game, and will sometimes attack each other, allowing players to make a fast getaway. The map graphics, Arabic-style typeface, music, and sound effects are quite good.

The problem with *Ali Baba* is the way it ends. After much exploration and fighting, you will discover the princess, and fight valiantly to return her to the king. When you manage to do so, the king thanks you, gives you some gold, and invites you to go back to amass more treasure. And that's it. It's a rather sudden let down, and it leaves players without the feeling they have mastered a difficult puzzle.



FOREST

ALI BABA

Ali Baba.

The documentation states that Ali Baba, himself a relatively weak, slow, and unskilled character in the adventure, can reach and rescue the princess unaided, and without once raising his sword to fight. As it still seems impossible to me after several hours of trying, this poses the challenge of a real puzzle. Realizing how much more fun it is to play this type of game with other people, I think it's a shame that the challenge evaporates so quickly in the multiplayer game.

Still, fans of role-playing games may really enjoy *Ali Baba*. It employs some nice concepts in its execution. For those protracted sessions, multiple games can be saved to disk.

Action Quest

Purist adventurers who feel arcade games are beneath them may wince at the inclusion of *Action Quest* in an article on adventure games. It is, in fact, a rather radical departure from the format of traditional adventure games. *Action Quest*, despite the uninspired title, deftly exhibits some of the qualities of the mapped adventure: it draws a map of current location, provides a running status report, and allows character movement to be input by joystick. Pull on the stick, and your character scrolls smoothly across the screen. Head for a portal, and a map of your new room location appears on the screen. But the game transcends the typical adventure from here on in.

SOFTWARE PROFILE

Name: Action Quest

Type: Arcade/Adventure

System: 16K Atari 400/800

Format: Cassette/disk

Language: Machine

Summary: Unique attempt to merge two types of games

Price: $29.95

Manufacturer:
JV Software
3090 Mark Avenue
Santa Clara, CA 95051

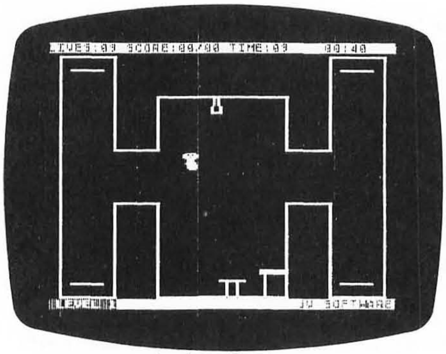
Action Quest is a one player arcade-style game within an adventure format—and it is addictive, well-paced, and fun.

The game is divided into five levels of six rooms each. If you complete all the required actions of each room, including gathering treasures, completing obstacle courses and traversing mazes, you may advance to the next, more advanced level of play.

You encounter numerous monsters, dodge bullets, run through rooms with walls closing in on two sides, and happen upon mystifying puzzles that must be solved. You carry a gun which you aim with the stick (this takes time to master), and while without it you wouldn't last too long, I still would not call this a "shoot-'em-up" type game. You have ten lives, which is not as generous as it sounds, considering the rate at which you expend them.

Each room has a name, which gives a clue concerning what you need to do to get through it. The only problem is that by the time you have read the name for the first time, you may already have been skewered by some bizarre creature.

You play against the clock as well as attempting to get through as many levels of the game as you can. This ensures you may still enjoy the game even after you have completed the adventure. I do not wish to instill the idea, however, that you will get all the way through *Action Quest* in any short amount of time. Mastery requires the acquisition of some formidable skills. At the end of the game your score is tallied, and you are assigned a rating on the basis of time used, lives expended, and treasures amassed.



Action Quest.

Perhaps it is a stretch to label *Action Quest* as an adventure game, but it is an exciting move in the right direction. The game is strong in one of the fundamental aspects of computer gaming: building toward a goal.

Though the sound is somewhat unsophisticated, the graphic animation is well-executed. Your character is an undulating ghost, and as remaining in one room for too long can be fatal, it begins to fade slowly when you enter a room. If the ghost disappears completely, it costs you a life. Sometimes you must shuttle between rooms quickly to avoid "suffocating" in this manner.

The author of *Action Quest* has indicated that a sequel with greater challenges, as well as more sophisticated graphics is in the works. I, for one, am looking forward to it. □

Cypher Bowl and Krazy Antics

Gridiron Action and Antics

Wayne Hixson and Sheldon Leemon

Cypher Bowl

"OK, Hixson. Zorn's hurt and out for the rest of the game. You're our man—now go out there and get 7!"

"Coach, you can count on me... Guys, its a 32 Up and Out. Largent, I'll be looking for you at the five. Break!"

A wild fantasy from the deranged mind of a short, slow, and (slightly) overweight sports nut? Not entirely—not with my Atari 800 and *Cypher Bowl,* an excellent two-player football game program by Bill Depew. Now we would-be jocks can step into the electronic shoes of a Jim Zorn, a Walter Payton, or a Jack Lambert to live our fondest fantasies in perfect safety. No injuries, unless you count acute "controller thumb," a malady now surpassing tennis elbow in popularity.

Cypher Bowl is attractively packaged in a sturdy, colorfully illustrated box. Both cassette and diskette versions are included (they are the same). The documentation is very good. The user manual explains the game clearly and gives good tips on playing techniques. Two playcards are included, laminated in plastic to withstand a lot of handling. Each playcard includes the offensive and defensive formations and plays. Sketches of each play show the patterns that the receivers, blockers, and defenders will run.

Once you have read the instructions and studied the plays, the game can begin. The program is self-booting and no cartridge is needed. The opening display is of the title, manufacturer, copyright notice, and a portion of the field. The crowd roars, and you're ready to go! Pressing any key turns on the standard display.

In the center of the screen are the field and the two five-man teams. You have a blimp's-eye view of the field, which runs vertically on the screen. The view is always centered on the ball. About 30 yards of the field are visible. The score, quarter, and time remaining are displayed on the top of the screen. On the bottom are the down, yards to go, and the time-outs remaining.

SOFTWARE PROFILE

Name: Cypher Bowl

Type: Football Simulation

System: Atari 400 or 800, 16K

Format: Disk or Cassette

Language: Machine language

Summary: Excellent combination of strategy and action

Price: $49.95

Manufacturer:

Artsci, Inc.
10432 Burbank Blvd.
N. Hollywood, CA 91601

The game is played in four simulated 8-minute quarters. There is no kick-off. The blue team starts with the ball on their 20-yard line, with the white team defending the top of the screen. Each player begins by selecting one of four formations from the playcards. On offense, you can spread your receivers, or play them in tight. The defense can put everyone up front, or drop up to three players back to play pass defense. After both have chosen, the teams move into position.

The players scrutinize each other's calls, then pick one of four possible plays allowed for the particular formation. Offensive possibilities range from quick openers to the bomb. The defense can opt for a strong pass, strong run, or balanced defense. What you choose depends on the formation your opponent unveils. For example, if you call a defense strong against the run and the offense deploys in a spread formation, you can still make the best of it by calling a zone defense to protect against the probable pass. However, your chances are poorer than if you had elected a strong pass defense formation to begin with.

Wayne E. Hixson, 115 NW 39th Street, Seattle, WA 98107.
Sheldon Leemon, 14400 Elm St., Oak Park, MI 48237.

This method of play calling is well thought out and superior to the other games I have played. Both players have options after they see the other's call, instead of the defense only.

Another nice touch is the way *Cypher Bowl* handles the 30-second clock. There is no delay of game penalty, but the clock is automatically stopped after 30 seconds until the play commences. The *Cypher Bowl* clock also stops between quarters, for the two-minute warning, for time-outs called by the players, and on incomplete passes and out of bounds plays.

After selection is complete, play is initiated as the offensive player moves the joystick. Instantly, the scoreboard information disappears and your view of the field increases to fifty yards. This is especially nice on pass plays, as the receivers would soon run out of view otherwise. During play, the offensive player controls the quarterback or the receiver, whichever has the ball.

As the manual points out, it is easier if you visualize yourself as controlling the ball, with the player coming along for the ride. On defense, you control the middle linebacker. The remaining eight players are controlled by the computer, following the patterns shown on the playcard.

Think about that for a moment. A total of ten players, moving in individual patterns. How? Aren't there only four players in Atari Player/Missile graphics? Yes, but *Cypher Bowl* shows just what a good programmer can do with this system. In order to get more than four players, single players are moved to different screen locations between TV frames, every 1/60 second. The images alternate so fast that the eye can't discern the change, except for some minor flickering.

As a result of the individual control of each player, blocking, passing, and pass coverage patterns are exceptionally realistic. If you make the right call, your left end will take the right linebacker out of the play and leave a hole a truck could drive through. However, if your opponent outguesses you and fills that area, you will be lucky to get back to the line of scrimmage.

Cypher Bowl excels in its simulation of the passing game. This was also the hardest part of the game to learn. Not only do you have control of passing direction, you must also control distance. In the other games I have played, a thrown ball will travel indefinitely, until it hits a receiver or defender, or goes out of bounds. Any eligible receiver (offense or defense) in the path of the ball will catch it, whether 6 or 60 yards from the quarterback.

Cypher Bowl adds a third dimension —height of the ball above the ground. Now you can throw the ball over the head of the defender. Of course, this also enables you to overthrow your own man, which I have been able to do very consistently. A pass is launched by pressing the joystick button and pushing the stick toward the target. The distance is determined by how long you hold the button down.

The height of the ball cannot be shown on the screen, so sound is used. A rising tone indicates a rising ball, and vice-versa. Once thrown, you can control the direction of flight with the joystick to "fine tune" it to the receiver. I think that this is one weak point of the program. The ball is too controllable. You can start it toward one sideline and then steer it clear across the field, or even reverse it back toward the quarterback. The magnitude of control should be reduced to a little nudging.

Another superior feature of *Cypher Bowl* is the option to throw to either of two receivers on most plays. You also have some control of the receivers on pass plays. Once the ball starts its downward flight, pressing the joystick button causes the receivers to break off their patterns and move back toward the ball.

As you can imagine, orchestrating all this activity in the period of about two seconds requires a lot of practice, but what a feeling when you float the ball over an onrushing linebacker to the tight end cutting back in front of the safety. A caution—there is only a five to seven yard window in which the receiver can catch the ball. If you overthrow, the defender is likely to get it.

You might think all this control would make an accomplished player unstoppable. Not so! This game provides a few tricks for the defense as well. If your defensive linemen get within a few yards of the quarterback before he throws, the ball will be batted down. Once the ball is in the air, you can make your defensive backs cut toward it by pressing the joystick button. All in all, the offensive/defensive balance is good.

The kicking game is good. The ball is kicked by pressing the button. Instead of going a random distance, the longer you wait before you press the button, the farther the ball will go. A split second too long, though, and it will be blocked. There is no difference between a field goal and a punt. If the ball goes between the uprights, it's worth 3 points.

I have played *Cypher Bowl* for over 30 hours now, and the more I play, the more I like it. The realism is a step above the other games I have played. The graphics, in spite of the lack of detail, are quite good. Player/Missile graphics, fine scrolling, and mixed modes are used very effectively. The animation is both smooth and fast.

The playability is good, and it's not an easy game to master. I'm still below 50% in the passing game, but I'm getting better. I think it is this continuing challenge in any game that keeps you playing it, along with the fun.

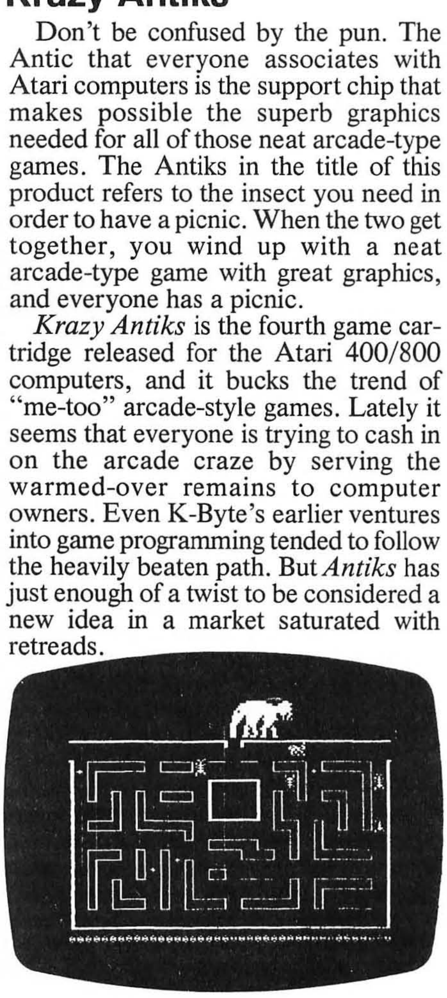
In summary, this is a worthwhile game. If you're a "stats junkie," it probably won't be your cup of tea, but if you like a sports game that makes you think and participate, I believe you'll love this one.

Now, guys, how about a solitaire version? I have a hard time finding opponents during my normal game-playing hours. —*WH*

Krazy Antiks

Don't be confused by the pun. The Antic that everyone associates with Atari computers is the support chip that makes possible the superb graphics needed for all of those neat arcade-type games. The Antiks in the title of this product refers to the insect you need in order to have a picnic. When the two get together, you wind up with a neat arcade-type game with great graphics, and everyone has a picnic.

Krazy Antiks is the fourth game cartridge released for the Atari 400/800 computers, and it bucks the trend of "me-too" arcade-style games. Lately it seems that everyone is trying to cash in on the arcade craze by serving the warmed-over remains to computer owners. Even K-Byte's earlier ventures into game programming tended to follow the heavily beaten path. But *Antiks* has just enough of a twist to be considered a new idea in a market saturated with retreads.



I must concede that the locale of the action is nothing novel—the ant hill in question strongly resembles the type of maze used in any number of games spawned by that prolific procreator, Pac-Man. But the scenario is a fresh one.

You play the role of the White Ant, and your purpose is one familiar to students of biology—to perpetuate the species. You start the game with about 30 eggs, which represent your capacity to reproduce, at the bottom of the screen.

Arrayed against you are several adversaries. First, one ant each of the four basic ant types—yellow, blue, green, and red—circulate around the maze, trying to devour you. Another natural enemy is the dreaded anteater, who strolls into the picture every so often and sticks his tongue into the anthill, sucking up friend and foe, ant and egg alike. Finally, periodically a rain shower turns the lower part of the anthill into a disaster area, minus the federal aid.

With the odds against her, the lone ant has little chance for survival. Fortunately, if she can find a safe place in the maze in which to lay an egg where it will not be eaten by another ant, after she is gone, the egg will hatch, and another white ant will take her place. Moreover, she has a weapon she can use.

The other ants are busy laying eggs also, and when she eats one of theirs, she begins to glow, letting you know that the next egg she lays will be deadly to the other ants, if laid directly in their path. At each level, play continues until the white ant is killed, without leaving any eggs in the maze, or until all four of the other ants have been killed without surviving offspring.

If the latter occurs, the game proceeds to the next level, and four new enemy ants come marching in to the tune of "When Johnny comes Marching Home" (which some like to think of as "The Ants Go Marching Two by Two").

Each maze has 99 levels of difficulty. If that fails to provide enough variation, there are six different maze configurations to try out.

Krazy Antiks rates a high score for playability. Even an experienced player can get caught early on by a freak accident, which inevitably leads to "just one more" game. There is a pause option, for those disturbed by the inconsiderate intrusions of friends and family. Unfortunately, there is no multi-player option. But if you don't mind going it alone, you'll bless the day when ant met Antic.

—*SL*

SOFTWARE PROFILE

Name: K-razy Antiks

Type: Arcade

System: Atari 400/800 16K

Format: ROM cartridge

Language: Machine

Summary: Puts ants in your pants

Price: $39.95

Manufacturer:

CBS Software
Columbia Group, CBS Inc.
Hagerstown, MD 21740

Not Just Fun in Games

John Anderson

Of all microcomputer inamorati, Atari owners probably take game programs most seriously. They are jaded; it's tough impressing the crowd for whom *Star Raiders* is invariably among the first programs ever booted. They expect more. Their machines, after all, were designed by games experts and exhibit advanced gaming capabilities. These capabilities are only now being fully explored, and are evolving, by leaps and bounds, into an art form.

Games offer an interactive and involving means by which to demonstrate strides in color graphics and sound synthesis. In the process, they afford a great deal of creative freedom to the programmer, not to mention hours of fun for the player. Still, a majority of Atari owners do not consider games mere frivolity. A swiftly growing market attests to this.

This has led to at least two identifiable results: an explosion of third-party software, some showing real promise, and translations of first-rate Apple programs for the Atari. We shall examine several of these here.

John Anderson is an associate editor for *Creative Computing* magazine.

Protector

I should provide a little background concerning this program, as it has an interesting past. Toward the end of last year, I reviewed a version of *Protector* written by Mike Potter and released by Crystal Software. The review, which appeared in another magazine, took the program to task for a number of flaws, "quirky bugs," and disappointing features. This was unfortunately true of the Crystalware version, and ruined an otherwise promising game. It was mysterious to me why an inspired program employing sophisticated techniques should be released in such a state.

The reason surfaced early this year, with the release of *Protector* from Synapse Software, Mr. Potter's new employer. It seems that when Mr. Potter left, his old company decided to market his as-yet-unfinished program. The Synapse version, I'm happy to report, not only corrects all the faults of the earlier version, but includes several new features, and to top it off, costs less than its predecessor. Needless to say, steer for Synapse *Protector* and away from any other.

The game is one of the most polished efforts I have seen from a third party source. It is exceptionally dramatic in its graphics and sound effects, and the animation is mirror smooth.

A great deal goes on in *Protector*, and mastery of the game requires a substantial amount of time. The game is roughly

SOFTWARE PROFILE

Name: Protector

Type: Arcade Game

System: Atari 400/800 32K

Format: Cassette or Disk

Language: Machine

Summary: Compelling and Addictive

Price: $29.95

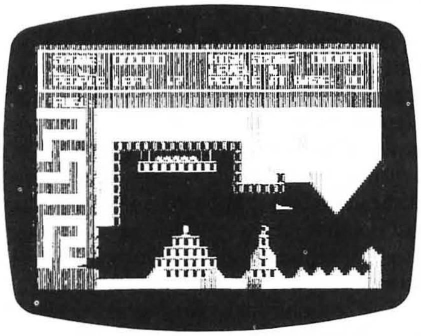
Manufacturer:

Synapse Software
820 Coventry Rd.
Kensington, CA 94707

modeled after the arcade game *Defender*. As the pilot of your rocket fighter, you encounter pulse-trackers, meteoroids, laser traps, a volcano, an evil alien ship, and 18 people in desperate need of your help. You must maneuver your ship so as to airlift these people from their beleaguered city to the City of New Hope, and from there to safety in an underground fortress.

You must act before they are heartlessly dropped into the volcano by the tractor beam of the alien ship, and before the volcano erupts and destroys the City of New Hope. You must also watch the fuel tank—and sometimes face the decision to refuel or to save some lives at the cost of your own. You can not always do both.

By far the best thing about the game is the horizontally scrolling terrain graphics. The overall goal is to create a "microworld"—a fantasyland one screen high by four or five screens long. Fine scrolling and player/missile techniques are employed to very pleasing effect. For demonstration purposes alone, this program is worthwhile.



Protector.

Sound effects add much to the illusion, and the title music is quite good.

The feeling of flight is accentuated as you dive to the rescue. Time ticks off as the indestructible alien saucer beams the victims up. Pulse trackers nudge dangerously close. Careful when you return fire: their favorite trick is to get you to hit innocent bystanders.

If you get all the remaining people to the City of New Hope, you can then move them through the laser field toward your goal. You must then watch for laser bases and meteoroids. When fuel runs low you must return to base to refuel. Docking can be a tricky and sometimes fatal task.

The game is paced into six levels of difficulty, graduated to present more aggressive aliens and more complex architecture through which to navigate. The merest graze of scenery, pulse-tracker, meteoroid, laser fire, or tractor beam, and you go down in a dizzying spin. An ambulance shoots out immediately to drag you away—what's left of you, that is. Better luck with your next ship.

I have very few reservations concerning *Protector*. As soon as a level loses its challenge, you may advance to the challenge of a new level. The highest level is very tough indeed. You may get a little tired of hitting things after a while, but after all, that's your own fault, right? Next time, be more careful.

Chicken

Mr. Potter has also created a children's game which will keep many adults busy after the kids have been tucked in. *Chicken* may be played with a joystick, but the responsiveness of a paddle is recommended to really rack up a score. Conceptually close to the arcade game *Avalanche*, the object is as follows: you, as chicken, must catch in your cart all the eggs dropped by a fox scampering across the top of the screen.

If you miss an egg, it hits the ground and cracks, and a peeping chick appears. As a chicken, you must fight back the instinctual urge to sit on the eggs you drop—an understandable but annoying habit. The trigger allows you a fluttering leap over the chicks in your quest to catch more eggs.

This may sound somewhat bizarre, and in fact it is. But it is also guaranteed to bring a smile to your face as well as to the kids' faces. As rounds progress, the action becomes more and more frenetic, with laughter as result. A recent competition among adults playtesting caused a bout of hysteria. It felt very good.

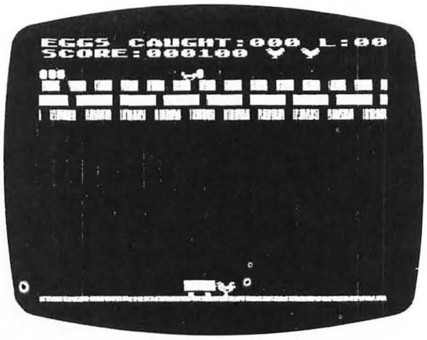
If you do plop down on a chick, a huffy farmer strides across the screen and gives you the boot. The addiction level is high, and the game is refreshingly violence-free.

SOFTWARE PROFILE

Name: Chicken
Type: Arcade Game
System: Atari 400/800 16K
Format: Cassette or Disk
Language: Machine
Summary: Silly, but fun—kids will love it
Price: $29.95
Manufacturer:
Synapse Software
820 Coventry Rd.
Kensington, CA 94707

By the time you have caught 40 or 50 eggs, your reasoning powers are on the wane. Ever hear the term "twitch game?" This game may be its namesake.

There is a dumb problem with *Chicken*, but it is worth mentioning. In an effort to make the game playable with stick or paddle, only one controller port is used.



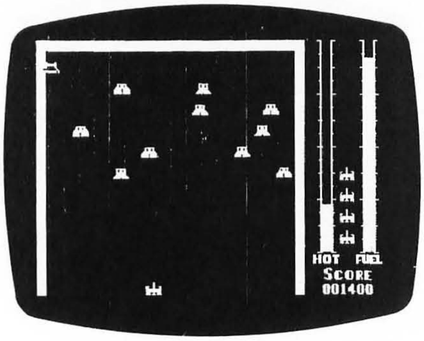
Chicken.

This means you must pass one paddle between players. This is by no means a major complaint, it just makes the game a bit less than it might be. The graphics and sound in *Chicken*, like those of *Protector*, are superlative, complete with barnyard music, the plop of dropped eggs, the peeping of chicks, and the fluttering of wings in a futile stab at flight. I've had an opportunity to learn about Mr. Potter's latest work, now in progress. I've promised not to spill a word, but I will tell you it sounds incredible.

Threshold

If I can get my wife to stop playing *Chicken*, I'm likely to take another shot at *Threshold*, which probably stacks up as the best "Galaxian-style" invaders game to date for the Atari. Atari owners will happily note that *Threshold* has been translated from the Apple, and that On-Line Systems is in the process of translating many of its popular Apple programs for the Atari. Also, new translations of best-selling Apple games are now available: *Apple Panic, Raster Blaster*, and *Crossfire*, to name a few. The translations of these games are at least as good, if not better, than their original versions. *Threshold* uses player/missile graphics, and *Raster Blaster* makes use of multi-channel sound.

Threshold is in the venerable tradition of laser-fire space wars (kill, kill, kill!) and it is superb. The alien waves in this game are ever-changing and wonderfully despicable. Your ships are armed with lasers and hyperwarp drivers that can temporarily slow down time, giving you a better chance to target the enemy. Your arsenal has limitations, however. The lasers can overheat and will shut themselves down until sufficiently cooled. You may invoke hyperwarp only once per ship, and each ship has a limited fuel supply. As for maneuverability, have you ever had the misfortune to be driving a power-steered vehicle that stalls while you're driving? That's the way the stick feels in *Threshold*.



Threshold.

The line between utter frustration and total addiction is a thin one, and this game rides it well. The game is hard to play but you can improve a little with every game. Aliens swoop down at your ship from the top of the screen, and each wave has its own character, its own "look." Some fly in jittery formation, others billow like a flag in the breeze. Your natural inclination to gape at them will prove fatal unless curbed. Discipline is called for in order to concentrate not on their grotesque beauty, but rather on their ability to destroy.

SOFTWARE PROFILE

Name: Threshold
Type: Arcade Game
System: Atari 400/800 40K
Format: Disk
Language: Machine
Summary: Best alien shoot-out to date
Price: $39.95
Manufacturer:
On-Line Systems
36575 Mudge Ranch Road
Coarsegold, CA 93614

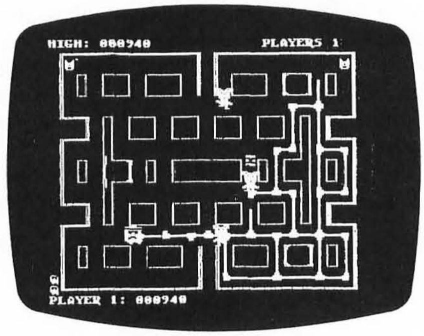
If you manage to survive a number of successive waves, you dock with the mother ship, which is rendered with the humor of a Saul Steinberg cartoon. Here you are refueled while a new set of nefarious alien waves are read from the disk. I have managed to live through two sets so far, and have yet to reach "the last wave." The documentation promises that when you get there, you'll know it.

You can choose to play with or without a moving star background (which makes it much harder to see enemy fire). You can also choose a horrific advanced level.

Threshold will obsess you for some time. Because the aliens change throughout the game, you're primed to withstand at least "one more wave *this* time." Though my wife abhors "shoot-'em-ups," even she spent a while with this one. After quite some time, I still have no reservations about *Threshold*.

Mouskattack

Mouskattack is a maze game that has a personality all its own. It moves beyond John Harris's earlier creation, *Jaw Breaker*, which set a standard for quality in Atari game animation. If maze games appeal to you, so will *Mouskattack*.



Mouskattack.

The game has several unique facets. Rather than "eating" as you traverse the maze, you are a plumber, laying pipe as you go. You are zealously pursued by a group of multi-colored rodents whose goal is to snuff you out. They have you on the run, so even after you have traversed the entire maze, some of the pipes may need to be reworked. Your only assistance in the completion of this task consists of a couple of traps, which don't hold rats for too long, and a couple of cats, who are too scared to do much more than discourage them a bit.

Mouskattack is *tough*. I don't care how experienced you are at any other kind of maze game—this one will pose a challenge. In fact, familiarity with other maze games may actually be a handicap! *Mouskattack* requires an entirely fresh approach.

The ultimately disappointing thing about many maze games is that the player can master rote winning patterns. Because the "enemy" follows the same patterns every game, routes can be learned which will work each and every time to avoid confrontation. In *Jaw Breaker*, Harris foiled the possibility of rote patterns by making the "enemy" much less prédictable. In *Mouskattack*, he provides a new element—strategic opportunity.

SOFTWARE PROFILE

Name: Mouskattack
Type: Arcade
System: Atari 400/800 32K
Format: Disk
Language: Machine
Summary: Another maze game, but can make your nose twitch
Price: $39.95
Manufacturer:
On-Line Systems
36575 Mudge Ranch Road
Coarsegold, CA 93614

Traps and cats can be picked up and moved in the course of your travels through the maze. This capability allows experimentation leading to strategic configurations. This is much more engaging than beating *Pac Man* with maps. The option makes you feel that more is involved than just conditioning and reflex action (though those qualities will certainly help your score).

The animation is very well executed, though lacking the inspiration and sparkle of *Jaw Breaker*. There are some flourishes, however. When you are "tagged" by a rat you drop down the screen like a leaf in a fall breeze. Every so often a "super rat" appears (easy to spot—watch for the "S"on its chest). Super rats will eat your cats right out from under you, so you must act fast when you spot one. The music in *Mouskattack* is quite well done, but begins to seem a little long after 10 or 20 airings. It is compulsory, and so loses its appeal in short order.

Still, *Mouskattack* has a lot of staying power as a maze game. In addition to offering strategic potential, it offers a simultaneous two player game, wherein you play against rodent and opponent at the same time. Squeaking good fun! □

Deluxe Invaders and K-razy Shootout
Blast From the Past

John Anderson

Deluxe Invaders

Your story may well be the same. Space Invaders, the first "cult" arcade game, hooked you—you, who vehemently swore your quarters would never be in short supply. It was the drum beat that did it: the quickening pulse that glazed over your eyes and tightened every muscle in your arms as you furiously raced to kill the last row of flapping insects.

Those were the days. I remember when *Invaders* first became available on cassette for the Atari computer. Finally, something had arrived to knock *Star Raiders* off the tube for a while. *Invaders* was well-animated, colorful, addicting, hilarious. But it was disappointing in its distance from the coin-op arcade game. Gone were the barriers that afforded temporary shelter from the falling "worm rays"; missing was the pace and feel of the game that was its inspiration.

Well it's been a while in coming—quite a while, actually—but the real thing is finally here. The nostalgia warms my heart. *Deluxe Invaders* faithfully captures the look, spirit, and play of arcade Space Invaders. And it doesn't stop there.

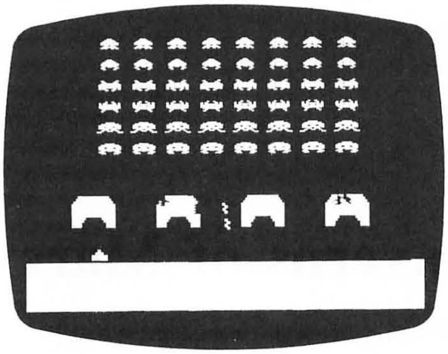
SOFTWARE PROFILE

Name: Deluxe Invaders
Type: Arcade game
System: Atari 400/800 16K
Format: Disk, ROM cartridge
Language: Machine
Summary: Finally a "genuine" Invaders implementation
Price: $34.95 disk, $40.95 ROM
Manufacturer:
Roklan Corporation
10600 Higgins Rd.
Rosemont, IL 60018

Deluxe Invaders retains the color, sound, and polish of the earlier Atari computer game, while remaining true to many of the features of the deluxe arcade game version. The barriers are back, as are the spinning "worm rays." Back also is the hypertensive pacing, and if you were into the game "back when," this game will go "click" when you start with it. Set aside some time.

There are nine levels of difficulty, including some where an insect results merely in its splitting into two baby insects. Other levels include mother ships that deposit new aliens on the board in play. Even the alien shapes are truer to the original game, as is the difficulty.

The difficulty levels are not too well documented, and only experimentation will flesh them out completely. The program does allow for a two player game, along the same lines as the coin-op.



Deluxe Invaders.

"What," you say? "Another Invaders game?" You're tired of Invaders games? I said the same thing when I first saw this package. I was wrong.

Roklan has some exciting plans for the Atari computer, including *Gorf* and *Wizard of Wor*. They are also planning a track-ball peripheral. If these products are up to the standard of *Deluxe Invaders*, we're in for a real treat.

K-razy Shootout

It's sometimes fun to trace the lineage of a game like *K-razy Shootout*. First there was *Star Wars*, with its stirring laser battles in the corridors of the Death Star. Audiences bobbed, weaved, and ducked in their seats as Luke, Han, and the Princess blasted their way through countless evil storm troopers.

Next there was the coin-op game Berzerk, pitting the arcader against evil 'droids closing in for the kill. The exciting "laser shoot'em-up" mood was evoked pretty accurately, constituting the appeal of the game. What's more, the game spoke, goading you, mocking you, teasing more quarters out of you.

Among a bevy of "laser motif" games for many systems, *K-razy Shootout* brings nearly all the excitement of the arcade game to the Atari computer. The only element that's missing is the speech. This is not to say that the Atari couldn't do it; it's simply not implemented here.

K-razy Shootout also bears the distinction of being the first ROM cartridge-based game from a third-party source. This necessarily adds to the cost of the package; but if you saw, enjoyed, and fondly recall the film *Star Wars*, you won't want to do without this program for long.

Your character runs through maze-like chambers, as 'droids close in from all directions. Using the joystick, you aim your laser, drawing a bead on them before they do the same to you. If you manage to clear a sector, you advance to the next. The action becomes increasingly furious, and you soon find yourself shooting from the hip, moving from sheer instinct, and totally addicted.

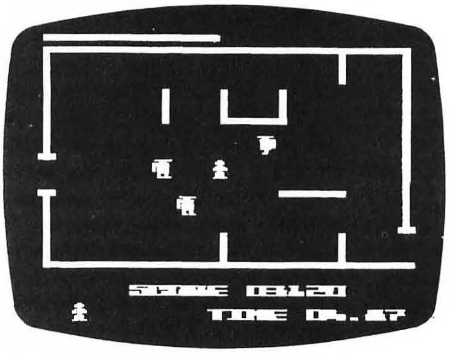
Scoring is dependent on several factors, including time, ammunition used, and 'droids' manner of demise: through hostile fire, collision, or shooting each other. In addition, you collect an extra player for every 10,000 points.

The only way you'll see sector four or beyond is through strategy. You'll discover that it's possible to get 'droids to collide or shoot each other—finding good cover is also imperative. Don't collide with a wall, though. That's as fatal as being hit by enemy fire.

SOFTWARE PROFILE

Name: K-razy Shootout
Type: Arcade game
System: Atari 400/800 8K
Format: ROM cartridge
Language: Machine
Summary: Addictive "shoot-em-up" game with classic roots
Price: $49.95
Manufacturer:
K-Byte
1705 Austin
Troy, MI 48099

The graphics, sound, and smooth animation in *K-razy Shootout* far outweigh its few negative points. The ranking system is screwy: you can progress from "Goon Class 1" to a higher score, which then is ranked back at "Goon Class 4."



K-razy Shootout.

John Anderson is an associate editor for *Creative Computing* magazine.

This frustrated our playtesters. The game can be paused, but only by pressing Control-1, as if you were in Atari Basic. A much more friendly option is using the space bar to pause, a function now standard on many games.

Still, *K-razy Shootout* is lots of fun, and has a great deal of staying power. If only it could talk. □

Dog Daze and Caverns of Mars

Dog Daze

We had just finished a picnic lunch of barbecued spareribs. I was walking the dog and as usual, when we passed a fire hydrant, he insisted upon investigating it, dragging me along on the end of the leash. David quickly grabbed a chewed bone, threw it, and hit the fire hydrant. "It's mine," he shouted gleefully.

David and I are not insane, just currently addicted to an APEX (Atari Program Exchange) game called *Dog Daze*.

SOFTWARE PROFILE

Name: Dog Daze

Type: Game

System: 32K Atari, disk drive,
2 joysticks or 8K Atari,
cassette drive, 2 joysticks

Format: Disk or cassette tape

Language: 6502 Assembly

Summary: Excellent game, lots of fun; highly recommended.

Price: $17.95

Manufacturer:
The Atari Program Exchange
P.O. Box 427
155 Moffatt Park Dr., B-1
Sunnyvale, CA 94086

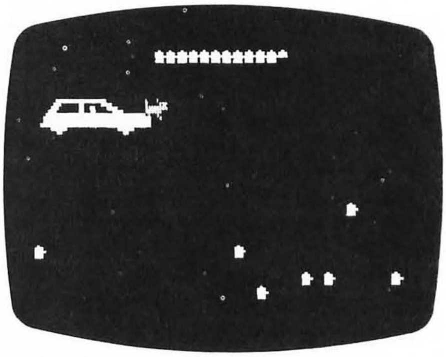
Now, I readily agree that the game concept which uses two dogs, fire hydrants, bones, and an occasional automobile doesn't sound as thrilling as being invaded by aliens. Furthermore, I will concede that the graphics are not as fancy as the ones in Centipede or PacMan. Then, you ask, just what is so good about *Dog Daze*? Why should I buy it? That's very simple to answer. The game is fun to play.

Dog Daze opens with the melody of "How Much Is That Doggy in the Window," then plots a play area and two dogs, each a different color. Along the top of the play area is a row of sixteen fire hydrants, eight of one color, and eight of the other. These fire hydrants keep score.

The object of *Dog Daze* is to get all the fire hydrants at the top of the play area your dog's color.

You maneuver your dog with the joystick, causing him to run vertically, horizontally, or diagonally across the playfield. Neutral fire hydrants (colored blue) appear on the playfield in random locations, and your dog must run to "claim" it. When the hydrant is claimed, it turns to the color of the dog that claims it.

You can claim a neutral hydrant one of two ways. One way is to run and touch each of the hydrants as they appear, thereby changing them to your color. The other strategy is to throw your bone at it by pressing the joystick button. If you hit it, you claim the hydrant, and get your bone back automatically. If you miss, you must retrieve your bone before you can throw it again.



In the meantime, your opponent is trying to do the exact same thing, making for a furious competition to be the first to claim the neutral hydrant.

There are several hazards to be avoided. If you pass too close to your opponent's hydrant, like all dogs, you must stop to sniff for a few seconds. While you are sniffing, your oponent's dog may be claiming all the neutral hydrants in sight.

An even more serious hazard is the car that periodically swerves across the playfield. It sounds a warning honk, but if you are in the path or stuck to a hydrant in its path, you may be hit and lose the game.

Scores are calculated based on two events: each time you claim a hydrant, one-half of one of the hydrants on top of the play area changes to your color and each time you run into the other dog's hydrant, one-half of one of your hydrants changes to his color.

The only options allowed in *Dog Daze* are to limit the length of the game, which is default sixteen minutes, and to handicap yourself by starting with fewer than eight fire hydrants of your color. This allows a skilled player to play with someone who is less skilled.

Sound is used quite imaginatively. A variety of bleeps, bonks, and various degrading noises (when you make mistakes) are generated. The dogs are animated quite nicely; the running motions are done very well. The author obviously took great care in designing his player tables.

Dog Daze is also one of the few two player games which allows both competitors to play at the same time. Most games use an "I go first, you go second" approach.

In summary, *Dog Daze* is an excellent game. It combines the capabilities of the Atari and an unusual game concept to achieve a truly enjoyable game.

Caverns of Mars

The *Caverns of Mars* arrived recently. I had heard rumors about this new Atari game, so I immediately sat down to play it and see what all the fuss was about.

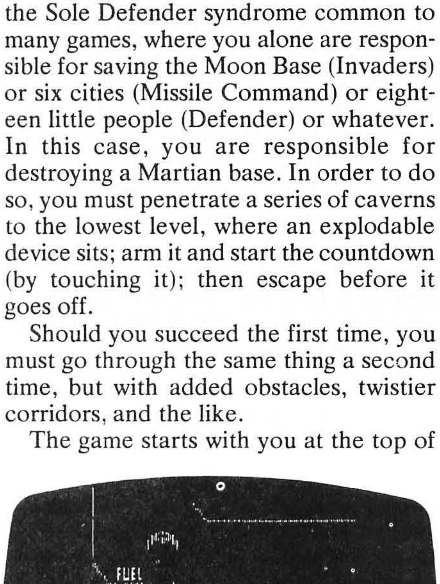
Four minutes later, I was hooked.

Four hours later, my wife dragged me away.

The plot is as follows (some of it is somewhat cliche, as it follows the lead of many, many other games.): First, there's the Sole Defender syndrome common to many games, where you alone are responsible for saving the Moon Base (Invaders) or six cities (Missile Command) or eighteen little people (Defender) or whatever. In this case, you are responsible for destroying a Martian base. In order to do so, you must penetrate a series of caverns to the lowest level, where an explodable device sits; arm it and start the countdown (by touching it); then escape before it goes off.

Should you succeed the first time, you must go through the same thing a second time, but with added obstacles, twistier corridors, and the like.

The game starts with you at the top of the cavern. It begins slowly scrolling up, so you move downward. By moving the joystick right-left you can maneuver from side to side (from a central position), and by moving it back and forth, you can increase or decrease your rate of descent.



SOFTWARE PROFILE

Name: Caverns of Mars
Type: Game
System: 24K Atari w/ Disk Drive
or 16K w/ tape, 1 joystick
Format: Disk or Tape
Language: 6502 Assembly
Summary: Excellently done game.
Price: $24.95
Manufacturer:
Atari, Inc.
1265 Borregas Ave.
Sunnyvale, CA 94086

If you pull the stick so that your ship moves upward on the screen, your ship matches the vertical speed of the caverns scrolling up past you. So your position relative to those caverns doesn't change; you have no vertical speed. But this can only last until your ship hits the top of the screen, at which point your relative velocity returns to normal. Similarly, if you move your ship downward, your velocity relative to the cavern walls is double that of no-motion.

This concept is what makes th *Caverns* so interesting, and difficult. If you don't move vertically, your rate of descent is constant and there are many places you must stop moving vertically to avoid running into the cavern walls (such as horizontal passages).

While you are descending through scenic Mars, you must destroy various installations. By pressing the joystick button, you launch two missiles downward from each side of your ship. If you hit a fuel canister (imaginatively labelled "FUEL") your fuel supply increases by 5 (of 100). If you hit other installations, you just plain destroy them. The idea is to wreak as much havoc as possible on the way down.

You can see only a limited section of the caverns. So you never know what's going to come next. You maneuver through a passageway twisting back and forth, and suddenly the screen is filled with Martian ships you must avoid, and try to blow up. But you must not collide with the ships or the wall.

If your first descent is successful you begin again. This time there are floating space mines, and force doors that open and close, and things begin to shoot back at you. Completely horizontal passages appear, requiring you to be ready for them and use nearly the full vertical screen's worth of maneuvering to get through. It gets harder and harder until you are destroyed, or somehow succeed in navigating all five caverns.

Technically, the game is excellently implemented. It's apparently done with remapped character graphics, letting the characters serve as the walls, ships, and so forth. Vertical scrolling is done smoothly and without flicker. The player tables for the ship are well laid out, and the missiles operate correctly. Sound is used well, with the usual explosions, rumbles, firing noises, and whatnot.

The Caverns of Mars has that indefinable "something" that makes it arcade-quality. Here's my best definition: When you lose in an arcade-quality game, you know why, and know how you could have done better, if you were just a little faster

Programming Precocity

Greg Christensen, author of *Caverns of Mars*, can't understand what all of the fuss is about. It certainly can't be the fact that he wrote an arcade action game for the Atari, or even the fact that it was good enough to win an "Atari Star" award. Perhaps it has more to do with the fact that he did so in less than two months, and despite the limitations of the Atari Assembler/Editor cartridge.

More likely it has something to do with the fact that he was 17 years old when he wrote it, after having a computer in the house for less than a year.

In addition, *Caverns of Mars* is the first program to make the transition from a package in APX, the Atari Program Exchange, to a part of Atari's main product line. Atari liked the game a lot, and invided Christensen to collaborate on the creation of a ROM cartridge version.

Young Mr. Christensen declined the offer. He has wisely decided to pursue an uninterrupted college education. Doubtless he has felt some pressure to surpass his feat, but has not succumbed. One cannot help but feel, however, that we may hear from him again.—*JJA* or if you hadn't have made that one mistake. Instead of the machine causing your destruction, it's your mistake that causes it. So, of course, you want to go back and try it again, and again, and get it right, until your fingers get cramps from holding the joystick, or until you're totally frustrated.

I recommend this game to anyone who likes fast-paced arcade games in the style of Asteroids or Missile Command and who is looking for a new challenge. □

Canyon Climber

Datasoft was among the first commercial third-party sources of Atari software, and the quality of their product line has remained consistently high. *Canyon Climber*, by Tim Ferris, sets a new standard for Datasoft, as well as one that challenges comparison.

Beginning with strains of Bach, *Canyon Climber* sets its own tense, yet humor-filled pace. The musical opening is superb; it is hard to tell whether you are listening to your computer or a cut from the album "Switched-On Bach." How Ferris manages the tone sustain is a mystery to me.

Suddenly the music vanishes, and your lone figure is left, clinging to a narrow canyon trail while dozens of surly, half-crazed billygoats seek to topple him from the precipice.

You are without weapons in your attempt to scale the many paths and ladders. Your only edge is a near ballet-like ability to leap into the air. If you time your leaps just right, you can hurdle goats on the fly. Your timing is crucial, of course; beginners will almost certainly earn a lot of horns in the keester.

The first task is to place explosive charges across a set of bridges spanning the canyon. Dodging oncoming goats from all sides, your fearless climber scales the sheer cliffs. And, upon reaching the detonator, you blow the bridges. This will hold the billygoats for a while.

However your problems are just beginning. The screen changes, and you find yourself at the foot of another set of cliffs.

Billygoats were child's play. Here you meet a very sedate group of Indians: they neither move nor make a sound. They simply and continuously shoot arrows at your face. Hope you've been practicing your pirouettes.

At a couple of these cliffs you will find a shield, which may help fend off a few arrows. Be careful though, because your shield may disappear at any time. Carrying it also makes ladder climbing tougher, as you cannot climb a ladder while carrying a shield.

If you are lucky enough to make it past the Indians, you are greeted by a final set of cliffs. You can see the top now. In the sky above, great birds hinder your progress by dropping, well, bricks in your path. The trail itself becomes quite tricky, as the way is broken by deep fissures. One misstep and you'll be goat feed by the time you hit bottom.

SOFTWARE PROFILE

Name: Canyon Climber

Type: Arcade game

System: Atari 400/800 16K

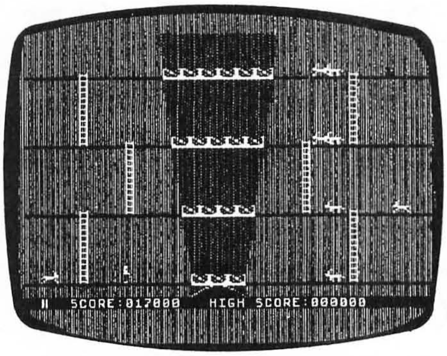
Format: Cassette/disk

Language: Machine

Summary: You'll want to gorge yourself

Price: $29.95

Manufacturer:
Datasoft Inc.
19519 Business Center Dr.
Northridge, CA 91324



Finally you reach the top, just long enough for a breath of blue sky and a bit of Bach before the head billygoat butts you right back down to where you started. This time the going will be even tougher.

Canyon Climber achieves a cartoon-like atmosphere in the rendering of its various screens, to very pleasing effect. Your figure has blond hair, and wears a blue shirt with jeans. It actually seems to throw a shadow on the canyon wall, as well. The animation is smooth and the colors superlative.

You will spend a while with *Canyon Climber*. It took me a couple of days just to reach the top on a regular basis. Now I have begun to work on my score.

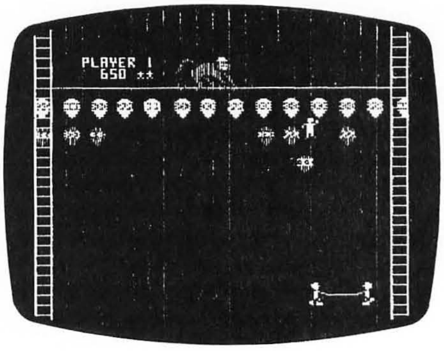
Clowns and Balloons

Several epochs ago, when I was a lowly undergraduate, arcade games were just beginning to use video screens. I remember an early one called Circus, and that it sat between Tank and Pong in the student union. Ah, those were the days.

"Oh, no," was my first thought when I loaded *Clowns and Balloons*, also from Datasoft. An exhumation of Circus: where is author Frank Cohen's respect for the moribund?

This report was exaggerated; I was dead wrong. This may very well be the most addictive game I have seen since *Threshold.*

In *Clowns and Balloons*, you manipulate a trampoline, shooting your player ever higher, as you try to break as many



balloons as you can in your trajectory across the screen. The concept is simple, but play is not. You must anticipate where to move that trampoline at all times. Otherwise, in the flick of an eye, your player will land in a headfirst heap on the floor.

Stone Age devotees of the black and white coin-op Circus will especially appreciate the sophistication of *Clowns and Balloons*. The trampoline is carried by two silver-haired clowns, whose outsized shoes scamper wildly as they run from side to side. The balloons spin and shimmer as they glide across the screen, and they do so in vibrant colors.

The music, as in *Canyon Climber*, is superb. Even after I landed on my head, I found myself humming along with it. Again, all factors work together to form an "atmosphere" about the game. It is as if it were a cartoon rather than a computer representation. It works very nicely.

It was easy enough for me to predict that my bevy of kid playsters would go nuts for *Clowns and Balloons*. They liked it nearly as much as I do. Fortunately, they belong to someone else, so I can play to my heart's content after they have gone home to eat dinner. Three levels of difficulty keep the action at a "breakneck" pace. □

SOFTWARE PROFILE

Name: Clowns and Balloons

Type: Arcade game

System: Atari 400/800 16K

Format: Cassette/disk

Language: Machine

Summary: Balloonatic adventure

Price: $29.95

Manufacturer:
Datasoft Inc.
19529 Business Center Dr.
Northridge, CA 91324

Pool 1.5

My initial reaction to the idea of pool on a computer was that it would be awfully hard to do well. Pool, I thought, would be too much of a physical game to run on a micro. IDSI has proven me wrong with the release of *Pool 1.5* for the Atari.

The first nice feature that *Pool 1.5* provides was revealed to me when I left my Basic cartridge in the computer and tried to load the program. A message appeared on the screen saying REMOVE CARTRIDGE. That was a refreshing change from BOOT ERROR. After removing Basic, the program loaded quickly.

The first screen shows the pool table and a prompt for the number of players (1-4). There is also a Demo mode which demonstrates the action of the balls during play. The computer does not play pool with people, but all of the four games can be played alone.

The players then enter their names, and decide which game is to be played. The game keeps track of whose turn it is by name and, in 8-Ball, will tell you who has solids and who has stripes. You can choose from Straight Pool, 8-Ball, Rotation or 9-Ball. The rules of each game are kept simple to allow for individual variations in play.

The only difference I found in the rules was in 9-Ball. The program returns the 9 ball to the table when it is sunk out of turn. When I play Uncle John in southern Maryland, the rule is that sinking the 9 ball using a proper combination is the end of the game. I have lost many quarters to Uncle John because of that rule, so I remember it well. The play of the other three games is pretty much the way I was taught.

The game is played with either the keyboard or the paddle controllers. The cue ball appears as a white ball on a red surface with a dotted line extending from it to a "ghost" ball which represents the point of impact for the cue. Rotating the paddle moves the ghost ball and provides coarse aiming at 128 different locations around the cue. Pressing the A key at this point allows fine aiming at a resolution of 32 positions to either side of the selected coarse position.

Aiming is only one of three parameters to be chosen, however. Pressing the space bar will bring you to the speed selection, where a 1 is a light tap and an 8 is an extremely hard shot. Pressing the space bar again switches down to the english selection. Using the paddle, you choose from top, bottom, center, left or right and combinations thereof. Pressing the paddle fire button shoots the ball. This can be done at any time, with the speed and english defaulting to your last selections.

SOFTWARE PROFILE

Name: Pool 1.5

Type: Billiard simulation

System: Atari 48K

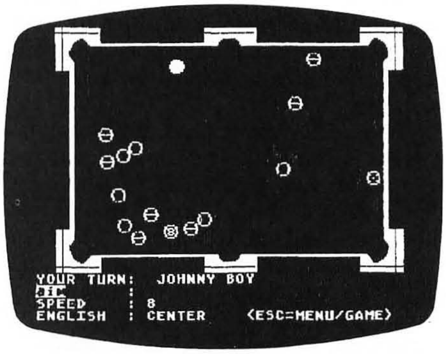
Format: Disk

Language: Machine

Summary: Excellent high resolution, real time simulation

Price: $34.95

Manufacturer:
IDSI
P.O. Box 1658
Las Cruces, NM 88004



```
YOUR TURN:  JOHNNY BOY
AIM       :
SPEED     : 8
ENGLISH   : CENTER     <ESC=MENU/GAME>
```

When the shot is off, the realism begins. The balls make a pleasant clicking sound as they hit each other, and a sunk ball makes kind of a gulping noise, as if it had been eaten. The physics of collision have been reproduced very well, and shots must be aimed and hit properly. In the case of a scratch, the cue is returned for positioning. The program questions the user if any balls sunk during the scratch are to be returned, allowing for individual tailoring of the rules.

There are several key-selected features in Pool 1.5. A favorite of mine is the Repeat Shot. Pressing the R key will restore the table to its last condition and allow you to change the angle or speed or english and try again. In several games with one of my cats, I've found this feature great for cheating. He doesn't know the difference.

The balls appear either as stripes and

solids, or with their numbers showing. The C (color) key allows the user to choose between these two options. The ESC key toggles between the game table and the menu/scoreboard. The scores for all players are kept here, and is updated each game so that a tournament of many games is possible. Other user-controlled functions include setting the table friction and the motion or speed at which the balls interact.

The keyboard commands are a little tough to master, and setting up a shot can require quite a few keystrokes. It takes several trips between the paddles and the keyboard before you can shoot. Although the high resolution part of the hi-res graphics is excellent, I find the uniform background and table color of red to be unattractive. Perhaps a later release of *Pool 1.5* will include a set-up feature to permit trick shots. But overall, the reproduction of the game of pool is accurate and fun. It is a relaxing and enjoyable game.

Nautilus

Mike Potter has done it again—this time in the guise of Captain Nemo. So much goes on in *Nautilus* it's hard to know just where to begin.

If you are familiar with *Protector* you will be reminded of it when playing *Nautilus*. Many facets of game play are similar, including a scrolling "microworld" several screens wide, and cities of steel and glass. Potter has developed an imaginative, storytelling style, and it is gaining in scope.

There are two independently scrolling screens in *Nautilus*. The top screen maps the progress of Colossus, the destroyer that constantly ferries repair teams across the microworld sea. It is armed with depth charges and heat seeking missiles, and can move at high speed. Among other dangers, the captain of the Colossus must remain wary of helicopter air attack.

The bottom screen maps the position of Nautilus, the malicious, energy-starved submarine. The Nautilus is armed with unlimited torpedoes, which are very handy—for the sea is filled with dangers. Besides the depth charges and smart missiles dispatched from Colossus, the deep is populated by limpet lurkers, dastardly and unrelenting smart mines. They lock on the course of the Nautilus, and maintain pursuit. It takes up to five direct torpedo hits to put one out of commision.

The goal of play for the commander of the *Nautilus* is to destroy underwater cities. In the cores of these cities reside the proto-pods which must be captured to replenish the voracious batteries of the sub.

Meanwhile, upstairs, the Colossus transports underwater repair crews to the rescue. On its way, it positions itself over the Nautilus, and unleashes a deadly mix of missiles and depth charges. It continues then to the western shore to drop off its crew. The crew will work its way eastward, repairing destroyed cities as it does.

SOFTWARE PROFILE

Name: Nautilus

Type: Arcade/strategy wargame

System: Atari 400/800 32K

Format: Cassette/disk

Language: Machine

Summary: Unique and engrossing action game

Price: $29.95

Manufacturer:
Synapse Software
820 Coventry Rd.
Kensington, CA 94707

If the Nautilus remains in proximity to a city under repair, it will be destroyed. It therefore becomes a priority to keep the Colossus from ever reaching the western shore.

The captain orders Nautilus to the surface, and steers it into shallows where there are no smart mines. Here it lies in ambush, in hopes of damaging Colossus enough to force it back to the eastern port.

The dual screen approach is unique, and allows the positions of both ships to be depicted simultaneously, even though they may be as many as five screens removed from one another. Sonar aboard each ship indicates the relative position of the other. When their screen locations coincide, the command console flashes red.

The ships can repair themselves as many times as necessary, but repairs cost precious time. The Nautilus must be careful not to so much as graze any solid surface—she goes down if she does. This makes navigation of the many underwater caverns a tricky business.

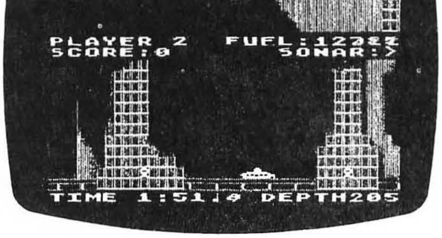
Nautilus can be played by two players, one at the helm of the Colossus, the other of Nautilus, or as a solitaire game, with the computer controlling the destroyer. Length of the game is selectable from three to nine minutes. There are nine skill levels, as well as the option to energize energy core transformers, gates throughout the sea, making the game extremely hazardous. This mode is not for beginners.There is a handicapping option as well.

Nautilus is a tour de force. The opening music is excellent, and hints at a context for the game—the tune is "Volga Boatmen."

The really appealing thing about the game, as in its predecessor *Protector*, is the creation of a microworld; in this case an undersea world, full of secret grottos and hidden dangers. The fine-scrolling graphics capabilities of the Atari are used to their fullest potential. *Nautilus* is another must from Synapse Software.

I do have a complaint, and though it is a small one, it is persistent. Allow me to appeal not only to Mr. Potter, but to *all* game designers with this plea: *please include a pause feature in your games*! Do it with the space bar, the escape key, CONTROL-1, SELECT; I don't care how, but please do it. It should be noted that the lack of such a feature becomes evident not only when the phone rings, but when it comes time for us to take pictures of a program for inclusion in a review such as this one.

Pause or no pause, *Nautilus* is addictive and a lot of fun. I recommend it very highly.



```
PLAYER 1   LEVEL: A10
SCORE:0        SONAR:<
PLAYER 2   FUEL:12288
SCORE:0        SONAR:>
TIME 1:51.4 DEPTH205
```

Shamus

In the August 1982 issue, I wrote about the burgeoning "arcade adventure" format for Atari games. I spoke specifically about *Action Quest*, a brutally tough but very compelling hybrid adventure, calling for hand-eye dexterity as well as adventure skills.

Shamus, also from Synapse Software, takes another stride in the development of the arcade adventure. Make sure you have no pressing appointments before becoming involved in a round of *Shamus*. Once you get going, you won't want to stop for a while.

The humorous feeling surrounding the game provides much of its appeal. Author William Mataga first sets the mood, with a grand rendition of the theme from the old Alfred Hitchcock show. The player is then thrust into a complex maze of 32 rooms, containing some very diabolical nemeses. As Shamus, the player must penetrate four levels of 32 rooms each, to finally destroy the Shadow in the heart of his lair.

SOFTWARE PROFILE

Name: Shamus

Type: Arcade adventure

System: Atari 400/800 16K

Format: Cassette/disk

Language: Machine

Summary: Another stride in "arcade adventuring"

Price: $29.95

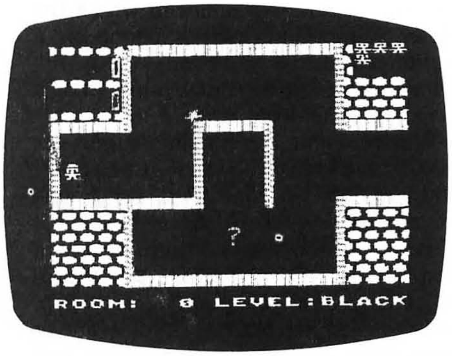
Manufacturer:
Synapse Software
820 Coventry Rd.
Kensington, CA 94707

Don't hold your breath waiting for the completion of this goal. It is bound to take you at least a month. You see, populating each room are the Shadow's henchmen: Whirling Drones, Robo-Droids, and Snap Jumpers. The sole pleasure in their lives is to keep you from getting near their leader. And they do a job of it. You are armed with Ion Shivs, and as your opponents are always prepared to fight to the death, the action is necessarily violent.

For those of you who have always wanted to wear a fedora as a character in an Atari game, this is your chance. The rakish lid is your most dashing feature. I'm not sure, but I think my hat has been shot through by more than one Robo-Droid blast. Take that, sweetheart...

Once in a while during your search you will encounter a pulsating question mark, the function of which is similar to "Chance" in Monopoly. By touching the punctuation mark you invite extra points and extra lives or ill fortune. I have found it hard to resist them in the long run.

To advance to a higher level, you must obtain the correct keys and unlock the correct portals. This calls not only for keen aim of your weapon, but knowledge of the labyrinthian layout of each maze. Secret passages abound, and it is quite easy to get lost. The bottom of the screen reads out a corresponding number for each room, and this is the only hint you get. I always seem to disorient myself right after unlocking a portal.



Your natural tendency is to shower attackers with ion fire. After a few games, however, you discover that fewer but better aimed shots will nearly always be a superior strategy. Keep cool, and if you find any bubbling flasks lying around, drain them: they will give you new life.

You will notice a couple of familiar tunes recurring throughout the game: one is from the old "Dragnet" series, and the other, if I remember correctly, is from "Get Smart."

Shamus is a very addictive detective game. It will remain in the front of your game software collection for some time, I guarantee it. Arcade adventuring is an emergent and promising gaming category, and this program underscores that fact.

Miner 2049'er

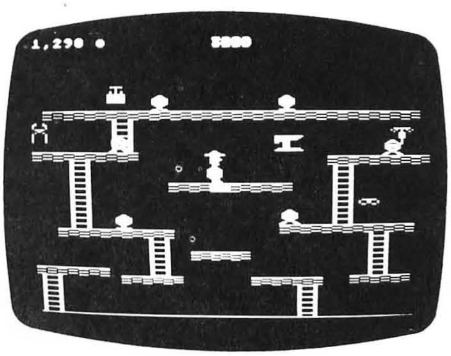
Atari Strikes Gold

Owen Linzmayer

Big Five, one of the leading software producers for the TRS-80, has introduced its first Atari 400/800 arcade game, *Miner 2049'er*. When I heard that the folks at Big Five were attempting to write an Atari program, I was a bit skeptical. After all, these guys know the TRS-80, not the Atari. After playing *Miner 2049'er*, I realized that my worry was for naught—*Miner 2049'er* promises to be one of the most popular Atari programs in any software library.

Miner 2049'er is written entirely in machine language by the president of Big Five, Bill Hogue. The whole program is crammed into a huge 16K ROM cartridge.

When you first see *Miner*, you can't help making comparisons between it and Donkey Kong. *Miner 2049'er* is similar to Nintendo's coin-op game in that they



Owen Linzmayer is a frequent contributor to *Creative Computing* magazine.

are both multi-level games in which the player jumps and scuttles about on a building framework. From there on, *Miner* proves to be much more than a variation of Donkey Kong.

Whether you are playing a one- or two-player game, your character, Bounty Bob, is controlled using one standard Atari joystick plugged into jack 1. To move Bob left, right, up, or down, simply point the joystick in the appropriate direction. To jump straight up, press the red fire button. If you want to jump from one place to another, you must be moving in the direction you want to jump when you press the button.

Whereas Donkey Kong has only four screens, *Miner* has a stupefying ten separate boards, each with a different scenario. In general, the object is to control Bounty Bob and "claim" all of the mine stations. Whenever you walk along sections of framework in the mine, the pieces under your feet will turn solid in color. To claim a station and advance to the next one, you must fill in every section of framework.

Bounty Bob can die in a number of ways, the most common of which is to run into a mutant organism. These creatures roam the mines in hopes of making your visit a short one. Falling too great a distance will also prove lethal, as will miscalculating a jump.

In addition to the deadly mutant creatures, every mine station has specific hazards that you must avoid (such as pulverizers, explosives, and slides). Scattered throughout the mine are various articles that have been lost by previous expeditions. To grab these objects, simply touch them. Points are awarded, and for a short time the mutants will turn green. While green, a mutant dies if you touch it.

As if dealing with all of these dangers is not enough, poor Bounty Bob must also race against time. Located at the top center of the screen is the "Miner Timer." If this timer reaches zero, Bob dies. Should you complete the station before time runs out, you are awarded the number of points remaining on the timer.

The limited sound effects are probably the weakest part of *Miner*. That is not to say that the audio is poor; it is just not up to the current standards for the Atari. Let's give a novice Atari programmer a little time to learn some of the better tricks for producing exhilarating sound effects.

The graphics in the game of *Miner* are detailed and very colorful. To guard against repetition, the color of the framework changes from station to station as well as from game to game. One of the most dazzling visual effects I have seen on the Atari is the animation of Bounty Bob dematerializing as he teleports from platform to platform using the elevators. *Miner* does not push the Atari to its full graphics potential, but it more than makes up for that in its limitless playability.

SOFTWARE PROFILE

Name: Miner 2049'er
Type: Arcade
System: Atari 400/800 16K
Format: ROM cartridge
Language: Assembly
Summary: Excellent multi-level game
Price: $49.95
Manufacturer:
Big Five Software
P.O. Box 9078-185
Van Nuys, CA 91409

As mentioned earlier, *Miner 2049'er* has ten independent mining stations (game boards). The first three sections are fairly easy to complete with practice, but the game gets much more difficult after that. Luckily, Bill sent me a copy of *Miner* that allowed me to "skip" to any station I wanted. If I hadn't received this special version of the program, I doubt that I would ever have seen what lies beyond the fifth station.

If you are skillful enough to accumulate a high score, you can add your name to the high score table. Unfortunately, the scores disappear when you pull the cartridge from the slot.

A multitude of stations and ever-increasing difficulty make *Miner* a game that is virtually impossible to master. *Miner 2049'er* is a great game—no doubt about it. After reviewing Bill Hogue's first Atari program, I can't wait to see what he comes up with next.

In late October, plans were being finalized with a variety of other manufacturers to produce versions of *Miner 2049'er* for all of the most popular home computers and video game systems. Look for adaptations of *Miner* for the following: Apple II, TI 99/4, IBM-PC, TRS-80, VIC-20, ColecoVision, Atari VCS and Atari 5200. □

Part IV
Disk Drive Tutorial

Atari DOS

A disk is a very complex piece of hardware. It is a mass storage device; one disk contains about twice the data that fit into the Atari's read-write-memory at one time. In addition, various functions must be supported; these include storage of disk files, random access of data, formatting and copying. All of these are controlled by the Disk Operating System; they are the "support" routines specific to the disk drive and are needed only to run it.

The Atari has a very sophisticated operating system, easily the best in the microcomputer market for the price. It is called the OS (not DOS—that is for the disk alone) and is physically located in the OS 10K ROM cartridge of the 800 and internal to the 400.

The Atari OS is very flexible and can do many unique things because it is "device independent." This means that any input/-output device-to-device communications are done not to a specific device, but to a "unit number." Whatever device is assigned to that unit number receives the instructions from the operating system.

For example, let's say that we have output going to unit number 2. An example might be a checkbook balance. Now if that unit number is assigned to device "TV screen," the output goes to the screen. If the unit number is assigned to the printer, the output goes to the printer. The output goes to the device to which the unit number is assigned.

This concept of device independent input/output is very consistent with the rest of the design philosophy of the Atari. For example, colors are not assigned directly; rather, a given screen image is drawn in a color register number. Whatever color is in that color register is then output to the screen.

The ability to reassign devices is extremely useful. Unfortunately, the workings of CIO (the Central Input/Output system) are a bit beyond the scope of this article.

Short detour (I warned you): Here's one bit of information for advanced users that is worth its weight in gold. In order to direct *all* output going to the TV screen to the printer, use:

C346 < A6,EE from the assembler/editor cartridge debugger.

Screen output can be restored with:
C346< A3,F6.

Let me cite an example. I was debugging a game that filled the screen with a graphics display. If output appeared on the screen, it would disturb that display—a rather common problem. By using the above modification, I got the debugging/trace output to appear on the printer instead, leaving the TV image "intact."

Back to Atari DOS. The DOS is a set of assembly language routines dedicated to running the disk drive. They load from disk any time the Atari is turned on with a disk turned on and connected. They are physically located in a file named DOS.SYS on the diskette.

Loading DOS

These routines are absolutely necessary to run the disk drive. If the file named DOS.SYS does not exist, is fouled up, or otherwise cannot be used, then the disk drive can't be used either. The Atari discovers on power-up whether a disk is present and attempts to load DOS.SYS from the plugged-in drive.

If the disk is blank, or anything else is wrong, the message BOOT ERROR appears, and the drive makes an awful "s-nn-aaa-rrr-kk" sound. Don't worry; the snark is the sound of the disk completely resetting its internal functions, the equivalent of "if at first you don't succeed..."

Okay, what happens after the DOS file loads? The Atari takes the disk routines and integrates them into its regular operating system. The routines to handle specific devices (such as the screen editor, cassette, or printer) now have the ability to handle the disk. (The DOS will go away whenever the Atari is turned off or crashed, incidentally.)

Regrettably, these routines occupy roughly 9000 bytes of memory, so you lose 9K for other uses. You need DOS to access the disk, so if you plan to use the disk at any time during the current power-on session, you must load DOS.SYS. This is something every Atari disk user has done—just turned the machine on, without disk, then tried to access the disk. When I did this last I typed a program in for half an hour, and typed SAVE—nope, couldn't do it.

(If you should get stuck this way, save the program to the cassette recorder, power up with DOS, and reload it from the cassette. The cassette handler is always in memory.)

Now for a little more relevant history. Atari has had several DOSes. The "first" DOS was dated 9/24/79, the date that shows up when DOS is typed. In this version, called DOS 1, the utility functions were integrated along with the regular operating system functions. When a user typed DOS, the utility functions were immediately run from memory, and the DOS menu popped up onscreen.

Well, this wasn't a winner, because these menu functions occupied about 3000 bytes of memory and were only needed when a specific disk utility function was required. DOS 1 also had other problems and bugs, so Atari came out with DOS 2.

In DOS 2 the utility functions occupied a separate file called DUP.SYS. When the user typed DOS, the utility routines were loaded from DUP.SYS off of the disk. They weren't in memory all the time.

There are some minor compatibility problems between DOS 1 and 2. Binary files won't work between them, as DOS 2 has a different "header" format, and copying is a problem.

Fortunately, most of the DOS 1 disks have disappeared, leaving users with an improved operating system which has eliminated many of the bugs. Alas, while Atari was working on the bugs, they "released" several preliminary DOS 2 versions, called DOS 2.4, 2.5, 2.8, and 2.S, all of which have bugs in them. Don't use them.

DOS 2.0S is the most bug-free version. (Should you find an older version of DOS, just re-write the DOS files after powering up from a 2.0S disk.)

One minor problem with the new DOS concerned where to load the DUP.SYS menu package in memory. The way it was set up, a user who went to DOS wiped out the lower 6K of memory, including any programs (such as Basic) stored there. The result? If you had a Basic program, went to DOS, and returned to the cartridge, your Basic program would be gone.

The solution Atari provided was MEM.-SAV. MEM.SAV is a special file created from the DOS menu. When you type DOS *and* a file named MEM.SAV exists, then the lower 6K of memory is moved to this file before the utility package (DUP.SYS) is read in. Hence, a copy of the lower 6000 bytes exists on disk. When DOS is left, the MEM.SAV file is read in, restoring memory to what it was. The process can be summed up as follows:

1. User enters a program into memory, including the 3000 bytes "shared" with DOS.
2. User types DOS.
3. Lower 6000 bytes of memory are copied to MEM.SAV on the disk.
4. The utility package (DUP.SYS) is read into the lower 6000 bytes, destroying the program data there.
5. User exits DOS.
6. MEM.SAV is read back in, restoring the lower 6000 bytes, and the user can pick up where he left off.

The process of reading and writing to disk is quite slow, as are all operations with the Atari drives. For this reason I rarely use MEM.SAV; I just save whatever

I'm doing to disk first, go to DOS, then recover it from disk later.

Another Sidetrack

Speaking of disk speed, new drives from other manufacturers are becoming available for the Atari. As a general rule, if the drive uses the serial I/O cable to attach to the Atari, it will run as slowly as the Atari disk; this cable is the bottleneck.

When the Atari writes something to disk, it normally re-reads the data written to disk immediately and compares what it finds there with what should have been written. This is a safety feature in case the disk doesn't write correctly. Alas, this slows down the disk drive to one write operation every 1/5 second, a very, very, slow speed. If you wish to cancel this read-after-write process, do this:

1. Power up with DOS 2 into Basic.
2. Type POKE 1913,80
3. Go to DOS and select H: Write DOS files.

The data at location 1913 determines what sort of write the disk drive does: read-after-write (87), or write alone (80). Next time you write to disk, you will notice an immediate increase in the write speed.

In all fairness, I have never once gotten the error message that means the read-after-write failed. Some of my associates have, but only on defective disk drives that gave numerous other errors. In my opinion, the write with no verify is the way to go, as disk operations are quite reliable. The time spent waiting for the Atari to verify data just isn't worth it.

The DOS Menu

Okay, so we have gone to DOS and are now in the DOS menu. Let's look it over.

The top line identifies the DOS and DOS 2.0S. The S means "single density" and refers to the amount of data written on a particular disk. Atari was going to offer a disk drive called the 815, which was a "double-density dual disk drive." For various depressing reasons the 815 was cancelled, so the double-density operating system, called DOS 2.0D, was never released.

Next, there's the copyright line. Then, the menu options begin. Let's take them in order.

A. Disk Directory: Data on Atari DOS disks is organized into individual files. These files have names of eight characters with an optional three-character extender; e.g. FILE.ABC, PROGRAM.BAS, and so on. Note that I said Atari DOS; there are other disk operating systems available which do not use Atari DOS. For instance, Forth doesn't generally use the Atari DOS at all, and an attempt to read the directory on a Forth disk is usually futile.

The directory is a list of the files, by name, which exist on the disk. Option "A" is used to read this list.

When you press A, the Atari asks, SEARCH SPEC, LIST FILE?

This means you can enter one of two items. The first is a "search specification." You can search for all files, in which case a list of everything on disk is produced, or for a specific group of files. This specific search is accomplished with "wild cards." A wild card is a special character which Atari DOS accepts as "any character." The character "?" is used for a wild card for an individual character, and "*" is used to indicate any characters from that position on. For instance, a search spec of *.* will find all files on the disk. *.BAS will find all files with the extension .BAS. JONES *.* will find any files whose first five characters are JONES. ??XYZ? will find AAXYZJ, ZZXYZD, and A1XYZR.

The second spec tells where to write the directory listing. Leaving it blank means write it to the TV screen, also known as "Device E:" (where the E stands for Editor). Here we get into the I/O system, which we have discussed previously. Devices on the Atari are identified by a letter and a colon. Here is a list of some of them:

K: Keyboard. Input only.
E: Screen Editor(TV). Input-Output.
S: Screen output. Output only.
C: Cassette unit. I/O.
P: Printer. Output only.
Dn:name.ext Disk drive #n, file name "name.ext."

The directory option asks where you want to write the listing. You can select any of these devices for output but the keyboard (K:).

You could use P: (printer), D:filename (some disk file), and so on. Note the power of the I/O system: you can write the listing anywhere, including devices or file names. For example, writing a listing to a printer is handy for a reference. Writing it to disk might be nice for a directory program. Writing it to the TV is good for quick lookups. This is a powerful unit.

The directory will then proceed. Physically the directory is located in the middle of the disk. This is because the Atari spends so much time looking at the directory that it was felt that the middle would be a good place; it is equidistant from everywhere else, saving lookup time.

B. Run Cartridge. This option transfers control to the plugged-in cartridge. If you don't have one, the Atari will figure it out and let you know it knows. This is how you get back to Basic, the ASM/EDIT cartridge, and others from DOS. Languages which are "disk based" (such as Microsoft Basic) do not use this option. There are different ways of getting to and from DOS using non-cartridge-based languages.

C. Copy file. This option allows you to copy from any device or disk file to any other device or disk file. It is extremely powerful.

For instance, if we copy from E:,D:TEST whatever we then type on the screen will be sent to the disk file TEST. (Exit using the Break key.) Bang, an instant crude word processor! We can copy directly from the screen to the printer (E:,P:), from the keyboard to the screen, etc. We can display a file on disk by using D:filename.ext,E:. (This includes Basic files, although they are stored in a crunched format and will look strange when listed.)

Finally, we can copy from disk to disk using this option: D:FROM,D:TO will copy all data in the disk file FROM to the file TO. Also, we can copy from disk to disk: D1:FROM,D2:TO will copy from FROM on disk #1 to TO on disk #2. The Atari can support up to four disk drives, D1-D4. The drive is identified by the two switches in the back of the disk unit; they can be set in four positions, and the position in which they are set determines the drive number of the disk.

Another short detour: If you can't get your system to "wake up," check these switches. The Atari will be looking for the disk #1 to get DOS.SYS from, and if no disk currently online has its switches set to 1, the Atari won't find it. This leads to all sorts of strange things. So especially if you have a multi-disk system, check this if you get weird errors.

You cannot use the Copy option on a single disk system to transfer files between separate diskettes. Use the 0 option to do this. 0 reads the entire file into memory then prompts you to change diskettes. Then, it writes the file out to disk.

COPY uses as much memory as possible as an intermediate storage place. If you copy a disk file to the screen, you will note that the entire file is read off disk before it begins copying to the screen. This is the nature of Atari I/O. You will also see that when copying to the printer, you must terminate the input operation before the output begins.

This causes a problem when MEM.SAV is used. When MEM.SAV is active, the Atari assumes that all memory outside the 6K bytes copied on disk is inviolate. On a copy, it will ask you whether to use the rest of memory to speed things up.

If you don't you will have a very slow copy, as only a small intermediate area in memory can be used. This also keeps memory intact for you to return to after you're done with DOS. Should you elect to allow Copy to use the rest of memory, MEM.SAV is invalidated and you lose whatever is on disk. The choice is yours. (The Atari will warn you that a "Yes" to its prompt will invalidate MEM.SAV.)

Warning: Files with ".SYS" as the extension will not copy using the wildcard options. While this doesn't really matter with DOS or DUP.SYS, as they may be written with the H option, it is critical with AUTORUN.SYS files. Be sure to force a copy of the AUTORUN.SYS filє if you copy a disk this way.

D. Delete File. This is an option to allow you to delete a file from disk. If you use a wildcard, you can get rid of a whole group of files. For example, to delete every file with an extension .ASM, use: *.ASM at the prompt.

Delete will ask you if you wish to delete each individual file by printing the file name, then asking DELETE? Y/N. If you don't want it to verify that you want the file deleted, add a /N at the end of the file specification. For instance, to delete all files with SAM as the first three characters and not get a prompt, use SAM*.*/N. The DOS will then delete everything it finds with those specifications without asking again if you really want to do it.

Delete *.*/N will erase an entire disk.

E. Rename file. This option allows you to rename a disk file. You enter the first file name, then the second. HERMAN, FRED will rename HERMAN to FRED on disk. Wild cards can also be used, but be careful.

This option also allows you to create two files with the same name—a significant problem. If you try to access the file by its name, the first occurrence will get priority, and you will have lost the second file for all practical purposes. But delete or rename will get both occurrences of the file, alas. What to do?

Try this. Turn up the TV sound. Rename the file something else, and listen. Immediately after hearing the first clunk of a disk write (not a beep, that's a disk read), pop the drive door open. This will prevent the Atari from renaming the second file, which would be the second clunk. Do this at your own risk—you could also trash the directory and lose the disk if your timing is wrong.

An alternative is to use a Disk Fixer program, such as the one available from APX, to alter the directory.

F. Lock file. A file that is locked may not be altered or deleted. This is a safety feature; I lock the editor and assembler files on my disks that have them. This prevents something like a wildcard delete from destroying them or something in DOS from accidentally modifying or destroying them.

G. Unlock file. The reverse of F.

H. Write DOS files. This option writes DOS.SYS and DUP.SYS on the current disk. (You are asked which drive number to write the files to.) Remember, you must have the DOS files on a disk to be able to power up using that disk, for the disk operating system must load at that time.

I generally use this option after modifying DOS (let's say with the "fast write" POKE) or after formatting a disk. By the way, old DOS files on the disk will be deleted. And in answer to a question I received, the DOS files do not need to be any place in particular on the disk. They can be put in any time.

I. Format disk. This option takes a new or used disk and completely blanks it out, putting "formatting information" onto the disk. It also sets up a blank directory and other information needed by the Atari to access the disk. And here we go on a short detour:

Fast Format Chips

A disk is laid out with the sectors in which data is saved in a particular order. As the disk spins at 290 rpm these sectors are accessed one by one. Now, depending on how the sectors are laid out, the Atari can access them more quickly. Atari has two popular sector layouts—the B and C layouts. The B layout is the original and is quite slow; there is a discernible pause between disk reads (beep—pause—beep—pause—beep, where each beep is one read.)

The C format is about 20% quicker than the B format, because the disk is laid out more efficiently. Disks that come from Atari use the C layout.

When you format a disk, the way your disk was set up at the factory determines whether it uses the B or C layout. Most drives today have the B layout, but all new drives shipped from Atari have the C layout. Thus, disks formatted on new drives (using the C layout) will do everything 20% more quickly than disks formatted on B drives.

By the way, if you reformat a disk, the new format will be the one laid out by your disk drive, so don't reformat Atari-formatted disks. Instead, if you want to delete old infomation from them, use Delete *.*.

A group of users in Chicago modified the B layout to what is called the Chicago layout. This layout is 30% quicker than the B format and indeed is 10% quicker than Atari's own C layout. However, a price is paid: the disks become rather sensitive.

Atari disk drives have difficulty maintaining a given rpm, which causes several problems, including lots of read-write errors. If you install the Chicago format, and your disk spins at more than 288 rpm, it will skip sectors, doing a complete spin between reads. This is quite slow and has a distinctive "Beepbeepbeep (pause) beepbeepbeep (pause)" sound. If you get this, check your disk.

One other thing about the Chicago chips is that they may be illegal. Atari copyrighted the B format in the ROMs used in the drive. It would annoy them considerably if users didn't buy the new C chips, complete with installation charge, but used the Chicago chips instead.

The legal question about copying the chips, then modifying them, is not one I would care to test. Yet many users have installed Chicago chips in their drives, and some groups even hold swap parties where hardware experts install Chicago chips into other people's drives. Someone with pretty good hardware knowledge and an EPROM copier is needed even to make the Chicago chips from the available instructions (which have shown up in many newsletters), so this choice may not even be available to you.

Yet another consideration is that the difference between the B and C chips available from Atari does not consist solely of the formatting change. The chips are much different, and supposedly other improvements have been incorporated into the C revision. You may be missing out on these improvements if you install a Chicago chip.

Another goodie installed by Atari on later drives is a piece of hardware called a "data separator." The story is this: Atari uses a floppy disk controller chip from Western Digital called the 1771. The 1771 is a fine chip, but has a weakness in clarifying data read from the disk, a process called data separation. Even the manufacturer's own literature tells the user not to rely on the internal data separation of the chip.

So what did Atari do? They didn't use an external separator. Result: bad disk reliability and lots of errors. Soon the more sophisticated users of Atari drives figured out the problem and began installing TRS-80 data separators in their Atari drives.

It seems that the makers of the TRS-80 had done the same thing (not used an external separator) and that TRS-80 disks had very poor reliability as a result. So outside companies began supplying data separators for the TRS-80. Since this machine also used the 1771 controller, the data separators for the TRS-80 fit the Atari.

I installed one some time ago and have been very pleased with the increase in reliability. The cost is $29.95 from one source, Percom, which now supplies kits for the Atari.

You need a soldering iron for two very minor solder touchups and a phillips head screwdriver to remove the cover of the machine. While the modification will violate the Atari warranty, it is worth it.

I recommend it to anyone who doesn't have the Atari data separator, which is everyone with a drive made before January 1, 1982. Percom can be reached at (214) 340-7081. You should call for new pricing and availability information.

Depending on your local dealer, parts availability, and other factors, you may be eligible for a deal whereby you send your drive in for installation of a C formatting chip and an Atari data separator and a general check-up. The Atari separator seems to be pretty good, so you may want to look into this option to upgrade your drive.

A late breaking rumor is that Atari has released yet another add-on board to help control the drive. I don't know whether this is true, but it sounds likely; drive rpm has caused many headaches.

DOS Menu Again

J. Duplicate Disk. (I know, you thought I'd never get back to the DOS menu. Right?) This option allows you to duplicate an Atari disk completely. What it does is read each sector from 1 to 720.

The user can either duplicate from drive to drive or with one drive by swapping disks. Use "1,1" at the prompt to duplicate a disk with one drive, and differing numbers to duplicate between drives.

Duplicate Disk is more or less identical to a copy using *.*. However, the disk duplication is complete, so errors in the disk will also be duplicated. Should you get an ERROR 14 or 164 on the disk, Duplicate Disk may not work, and you should copy individual files from disk to disk to recover what can be recovered. A discussion of sector chaining and what causes an Error 164 is beyond the scope of this article, but can be found in the April and May 1982 issues.

K. Binary Save. This is an option for the advanced user which saves a given area of memory to disk as a binary file. It is an assembly language entity used by the machine. Since this is a beginner's guide, and hexadecimal input is required, I'll leave it at that. See the DOS 2 manual for a lengthy, painful disussion of what happens.

L. Binary Load. This is an option to load a binary file from disk into memory and to execute it directly. Beginners may use it, although they may not understand what is going on. The Macro-Assembler/Editor is only accessible by loading it from a binary file, for instance. And Microsoft Basic is just another binary load file. (Think of a cartridge as a binary load frozen into the cartridge which appears in memory when you plug the cartridge in, and a disk load as data that appears in memory loaded from disk. This will give you an idea as to how the two relate.) And no, you can't copy a cartridge using the Binary Save option—Atari DOS checks for this to prevent people pirating the cartridges.

M. Run at address. Again, this is an advanced-user-only option. It enables DOS to jump directly into a program loaded in memory. It is handy for advanced users who want to run programs without a cartridge, but not so helpful for beginners. Again, knowledge of hexadecimal is required.

N. Create MEM.SAV. This is used to create the initial MEM.SAV file. To eliminate it, use the Delete option. You cannot create MEM.SAV any other way, although a disk that is Duplicated will have the MEM.SAV on the new copy if the FROM disk had it.

O. Duplicate file. This is used to copy a file from one disk to another without using two drives. Wildcards can be used to copy an entire disk.

Disk drives are relatively high-speed mass storage devices. Alas, the 5 1/4" mechanisms represent a tradeoff between reliability and cost. The 8" drives, which are more reliable, also cost much, much more. Atari probably couldn't market an 8" drive for less than $900; so they went with the 5 1/4" mechanism and enabled many more to have disk drives. It was a good tradeoff.

Unfortunately, the way in which Atari designed their drives is developing into a controversy. The number one topic of conversation in many user's groups seems to be peeves about Atari disk drives. The drives are neither reliable nor fast—even compared to the rest of the industry. Apple disk drives, for example, run up to 20 times faster.

Something will undoubtedly be done; Atari has not been deaf to the complaints. For the moment, they have issued several patches to the drives—data separators, rpm fixes—but they may not be able to correct what might be simply a bad design.

See you next time! □

Atari Diskfile Tutorial — Part I

Jerry White

Many new computer owners are anxious to learn how to write their own useful programs. After reading the literature packed with the machine, the new owner is often overwhelmed. Realizing that one does not learn any programming language overnight, a seemingly endless period of trial and error usually follows. The "hacker" is often seen burning the midnight oil and arguing with a defenseless TV or monitor.

Jerry White, 18 Hickory Lane, Levittown, NY 11756.

If he perseveres long enough, reasonably simple programs are written. The new programmer is now ready for bigger and better things.

Assuming he has a disk drive, our "hacker" gains experience with DOS and the loading and saving of programs. Now he is ready to write a database program.

The datafile may consist of a simple list of record albums for a start, to be followed by the inevitable Personal Finance System. If you are at this point in your programming career, or think you might be in the near future, read on.

Start with something very simple. Don't try to write that financial package yet. There is much to learn first about file structure and I/O. I/O stands for Input/Output. Input is data being read by a program. Output is data being created by a program. A file consists of one or more records, and a record is an item within a file. Records may be broken down further into fields. We will be using simple records containing a single 20-character field as our record, and create a sample 10-record datafile.

To understand data processing techniques, it is often easier to grasp reality than it is to learn by reading. I have found that doing is the best way to learn, and that Atari Basic can be easy to understand if it is explained in English.

Atari Basic allows variable names of any length, plus REM or remark statements. Remarks or comments within a program help identify routines and explain exactly what the program is doing.

Meaningful variable names also make program reading much easier. For example, the sample Diskfile program uses the variable RECNUM to store the current total of records. RECNUM is an abbreviation I used to mean record number. So why didn't I use the variable RECORDNUMBER you ask? RECNUM is a compromise between that 12-letter name and the other extreme which could have been R.

The RECNUM variable is used often. The tradeoff is readability against the programmer's keystrokes and sometimes program efficiency. If R is used instead of RECORDNUMBER, and that variable is used ten times, using R saves 110 keystrokes. In a tutorial program such as this one, RECNUM is the acceptable compromise.

The Diskfile tutorial program demonstrates many of the common functions required in a simple database type program. By using the program and studying the program code, you will learn how datafiles may be handled in Atari Basic. Once you have entered the program and corrected any typing errors, run through each of the options beginning with number one.

It is important to understand the terminology used here. CREATE means just that. In this case it means create from

```
0 REM FILES (c) 1981 by Jerry White
1 REM ATARI DISKFILE TUTORIAL DEMO
2 REM
100 DIM DRIVE$(3),FILE$(12),DRIVEFILE$(15),RECORD$(10),ANSWER$(1)
110 DIM SECTOR(20),BYTE(20),DIRECTORY$(20):REM DIMENSION STRINGS AND ARRAYS
111 REM
120 GRAPHICS 0:POKE 82,2:POKE 83,39:REM CLEAR SCREEN AND SET MARGINS
130 POKE 201,5:REM SET PRINT TAB WIDTH TO 5 SPACES
140 ? :? "TYPE OPTION NUMBER THEN PRESS RETURN"
150 ? :? ,"(1) CREATE A DISK FILE":REM GOTO 1000
160 ? :? ,"(2) READ A DISK FILE":REM GOTO 2000
170 ? :? ,"(3) ADD TO A DISK FILE":REM GOTO 3000
180 ? :? ,"(4) UPDATE A DISK FILE":REM GOTO 4000
190 ? :? ,"(5) DISPLAY DISK DIRECTORY":REM GOTO 5000
200 ? :? ,"(6) END PROGRAM":REM GOTO 9140
210 ? :? ,"YOUR CHOICE";:GOSUB 7000
220 TRAP 8000:LINE=120:HIGHNUMBER=6:NUMBER=VAL(ANSWER$)
230 IF NUMBER<1 OR NUMBER>6 THEN GOTO 8000
240 ON NUMBER GOTO 1000,2000,3000,4000,5000,9140
250 REM
1000 LINE=6100:GOSUB 7100:TRAP 9100:GRAPHICS 0
1010 CLOSE #1:OPEN #1,8,0,DRIVEFILE$
1020 ? :? "CREATING ";DRIVEFILE$:? :RECORD$="1234567890"
1030 FOR DEMO=1 TO 10
1040 ? #1;RECORD$
1050 ? "WRITING RECORD NUMBER ";DEMO
1060 NEXT DEMO
1070 ? :? "10 RECORD DEMO FILE CREATED"
1080 ? :? "CLOSING ";DRIVEFILE$
1090 CLQSE #1
1100 GOTO 6100
1110 REM
2000 LINE=6100:GOSUB 7100:TRAP 9100:GRAPHICS 0
2010 CLOSE #1:OPEN #2,4,0,DRIVEFILE$:RECNUM=0:LINE=6100
2020 INPUT #2,RECORD$
2030 RECNUM=RECNUM+1
2040 ? "RECORD NUMBER ";RECNUM;
2050 ? ,RECORD$
2060 GOTO 2020
2070 REM
3000 LINE=3000:GOSUB 7100:TRAP 9100:GRAPHICS 0
3010 CLOSE #3:OPEN #3,9,0,DRIVEFILE$
3020 GRAPHICS 0:? :? ,"ADD RECORD(S) ROUTINE:"
3030 ? :? ,"ENTER 10 CHARACER RECORD"
3040 ? :? ,"OR JUST PRESS RETURN TO EXIT":? :GOSUB 6000
3050 RECLEN=LEN(RECORD$):IF RECLEN=0 THEN 3200
3060 IF RECLEN=10 THEN 3090
3070 FOR BLANK=RECLEN+1 TO 10:RECORD$(LEN(RECORD$)+1)=" ":NEXT BLANK
3090 PRINT #3;RECORD$
3100 ? :? "PRESS START TO ENTER ANOTHER RECORD"
3110 ? :? "PRESS OPTION FOR OTHER OPTIONS...";
3120 IF PEEK(53279)=6 THEN 3020
3130 IF PEEK(53279)=3 THEN 3200
3140 GOTO 3120
3200 ? :? :? ,"ADDING RECORD(S) TO DISK":CLOSE #3:GOTO 120
3210 REM
4000 LINE=4100:GOSUB 7100:TRAP 9100:GRAPHICS 0
4010 CLOSE #4:OPEN #4,12,0,DRIVEFILE$:LINE=4100
4020 ? :? ,,"CREATING INDEX":RECNUM=0
4030 NOTE #4,SECTOR,BYTE
4040 RECNUM=RECNUM+1
4050 SECTOR(RECNUM)=SECTOR:BYTE(RECNUM)=BYTE
4060 INPUT #4,RECORD$:? ," RECORD ";RECNUM,RECORD$
4070 ? ,"SECTOR=";SECTOR,"BYTE=";BYTE
4080 ? :GOTO 4030
4100 RECNUM=RECNUM-1
4110 ? :? "PRESS START TO UPDATE A RECORD"
4120 ? :? "PRESS OPTION FOR OTHER OPTIONS";
4130 IF PEEK(53279)=6 THEN 4200
4140 IF PEEK(53279)=3 THEN CLOSE #4:GOTO 120
4150 GOTO 4130
4200 GRAPHICS 0:REM RANDOM ACCESS RECORD UPDATE ROUTINE
4210 ? :? ,"DISKFILE CONTAINS ";RECNUM;" RECORDS"
4220 ? :? "ENTER RECORD NUMBER TO BE UPDATED";
4230 TRAP 4220:INPUT UPDATE:TRAP 40000
4240 UPDATE=INT(UPDATE):IF UPDATE<1 OR UPDATE>RECNUM THEN 4230
4250 POINT #4,SECTOR(UPDATE),BYTE(UPDATE)
4260 INPUT #4,RECORD$:? :? RECORD$
4270 ? :? "ENTER NEW RECORD #";UPDATE;:INPUT RECORD$
4280 RECLEN=LEN(RECORD$):IF RECLEN=10 THEN 4300
4290 FOR BLANK=RECLEN+1 TO 10:RECORD$(LEN(RECORD$)+1)=" ":NEXT BLANK
4300 POINT #4,SECTOR(UPDATE),BYTE(UPDATE)
4310 PRINT #4;RECORD$:? :? ,"RECORD HAS BEEN UPDATED"
4320 GOTO 4110
```

scratch. Note that the create routine actually begins at line 1000 and that line 1010 contains an OPEN command. The number 8 in that command means write only. If a file is opened using this variable, and a file with the exact same name is found on your diskette, the old file will be deleted automatically.

Using option two, a file is read from disk and displayed on the screen. This does not in any way alter the disk file.

Option three is used to ADD data to an existing disk file only. The term APPEND is often used in this case. In plain English, the term APPEND means, "add to the end of this file."

Option four is used to UPDATE the records of an existing file. This means you will alter, correct, or change a record. This procedure is a bit more complicated than the others since we do not know in advance which record the user may choose to update. The technique used in this demo program is known as Random Access Updating. An index consisting of SECTOR and BYTE locations is created and stored in an array. This gives us the exact spot at which each record begins.

Since we are using fixed length records of 20 characters each, we can read a specific record into a string, change it in the string, then rewrite the string onto the disk. This becomes a real time saver when many records must be updated in a large disk file.

Option five is used to READ and display a specific file called the DIRECTORY FILE. This DOS-generated file contains the table of contents of your diskette. This file is also known as the VTOC or Volume Table Of Contents.

```
4330 REM
5000 GRAPHICS 0:POKE 201,10:? :? ,"  DISK DIRECTORY":? :TRAP 9100
5010 CLOSE #5:OPEN #5,6,0,"D:*.*":REM OPEN DISK DIRECTORY FOR ALL ENTRIES
5020 LINE=6100
5030 INPUT #5,DIRECTORY$
5040 ? ,DIRECTORY$
5050 GOTO 5030
5060 REM
6000 RECORD$="":POKE 764,255:REM RECORD STRING AND LAST KEY PRESSED=NULL
6010 INPUT RECORD$:RETURN
6020 REM
6100 FOR FILE=1 TO 5:CLOSE #FILE:NEXT FILE:REM CLOSE ALL FILES
6110 POKE 201,5:? :? ,"PRESS RETURN FOR OPTIONS";
6120 GOSUB 7000:GOTO 120:REM PAUSE TO READ SCREEN THEN GO TO OPTIONS
6130 REM
7000 ANSWER$="":POKE 764,255:INPUT ANSWER$:RETURN :REM 1 CHARACTER INPUT
7010 REM
7100 GRAPHICS 0:REM DRIVE NUMBER AND FILENAME INPUT ROUTINE
7110 ? :? "TYPE DISK DRIVE NUMBER (1-4)";:HIGHNUMBER=4:GOSUB 7000
7120 LINE=7110:TRAP 8000:NUMBER=VAL(ANSWER$):TRAP 9100
7130 IF NUMBER<1 OR NUMBER>4 THEN 8000
7140 DRIVE$="D":DRIVE$(LEN(DRIVE$)+1)=ANSWER$
7150 DRIVE$(LEN(DRIVE$)+1)=":"
7200 ? :? "TYPE FILE NAME";:INPUT FILE$:IF LEN(FILE$)=0 THEN 7200
7210 DRIVEFILE$=DRIVE$
7220 DRIVEFILE$(LEN(DRIVEFILE$)+1)=FILE$:RETURN
7230 REM
8000 ? :? "PLEASE TYPE A NUMBER FROM 1 THRU ";HIGHNUMBER:REM ERROR ROUTINE
8010 GOSUB 9000:GOTO LINE:REM GO BACK TO LINE NUMBER (LINE)
9000 ? CHR$(253):REM RING ERROR BELL
9010 FOR COUNT=1 TO 300:NEXT COUNT:RETURN
9020 REM
9100 IF PEEK(195)=136 THEN GOTO LINE:REM ERROR WAS END OF FILE
9110 REM DISPLAY ERROR NUMBER AND LINE AT WHICH ERROR OCCURRED THEN END
9120 ? :? " ERROR ";PEEK(195);" AT LINE ";PEEK(186)+PEEK(187)*256
9130 LIST PEEK(186)+PEEK(187)*256:GOSUB 9000
9140 TRAP 40000:END :REM ELIMINATE ANY PREVIOUSLY SET TRAP AND END PROGRAM
```

For display only, this routine does the same thing as DOS option A.

Although some error trapping has been built in, many possible error conditions are not corrected or fully explained by this program. Error trapping and human engineering account for a great deal of planning and program code. This is not a cop out on my part. I plan to cover this subject in a future article. The point here is to provide an example of diskfile handling. Accounting for all possible errors could easily double the size of the program.

That's about it for now. I suggest you use my program as is, then experiment by making minor changes and noting the results. When you're ready to write your own diskfile handling program, feel free to use these routines. □

Atari Diskfile Tutorial — Part II

For those of you who have read and been entertained by the discussion of the Atari disk in the Basic Reference Manual, this article will let you in on what is really going on. It is a ground level look at how the disk works and what the Atari does with it.

The information is quite useful and lets you access the disk more efficiently in many applications, as well as understand how Basic and DOS work with the disk at the lowest level.

This article, believe it or not, presented its authors with a real moral dilemma. Why? The information presented here can be used to copy disks that without this information are uncopyable. (Mind you, this information is also freely available in the Atari O.S. manual, if you can understand it.)

There is a horrendous software piracy

problem in this country. Consider the effect on the record industry if every time a person bought a cassette tape, he made ten cassette copies of it and gave them to his friends. This is exactly what is happening in the computer industry. (Indeed, the record companies have come out strongly against the sales of blank tapes recently and will no longer support ads for those products, for just this reason).

Immediately after a new program appears on the market, it is copied. The copier then distributes copies to friends, trades them for copies of other programs, and so on. And the writer of the software receives no royalties from the copies of his program.

There are ways of making disks uncopyable. This seems like an ideal solution until you realize that a diskette can be destroyed. A phone can ring near it and erase it; a cat can use it for a scratching post (our cat, Atari, tried this trick once), and so on.

If a user is depending on the disk, and it quits working, then he is in trouble. For this reason the practice of "backing up," or making multiple copies of a disk, was started. Too many users had lost their only copies of badly needed material.

It's a good idea; I keep a minimum of two copies of everything on disk, so if I lose one, I still have what I need. And it has paid off; many times I have had to go to the backup copy when the original died for some reason. I don't know why, but I seem to lose an amazing number of disk files—error 144, error 164, and so on.

And if a manufacturer has made his disk uncopyable, you can't back it up. If the disk becomes unusable, you're out of luck.

The software industry has been debating this problem for ages. The Apple, in particular, has been the subject of controversy, with some manufacturers selling programs written for the express purpose of copying "copy-protected" disks. In their literature they describe how backup copies are necessary, and thereby justify the purchase and use of their programs. But they know, and everyone else knows, that these programs are used to rip off an enormous amount of copy-protected software.

Copy Programs

One program in particular, *Locksmith*, gave me a good laugh. *Locksmith* is an Apple program written specifically to copy disks that are copy-protected, and has sections to handle all of the latest protection techniques. It has caused a great deal of controversy.

In the *Locksmith* manual, the writers explain the need for backup copies, how disks can be erased, and so on, and condemn manufacturers for copy-protecting disks for this reason. Then, after going to great lengths to point out that *Locksmith* is for users to make backup copies of programs, the manual points out that *Locksmith* cannot make copies of itself.

In other words, they know what's going on—who's kidding whom? They even maintain a database on The Source which tells how to copy many of the copy-protected Apple programs using *Locksmith.*

Justice has been served, and there are now many, many bootleg copies of *Locksmith* floating around. As Scott Adams might say, "Yoho, and Jolly Roger."

Well, Atari users are in the same boat; things just haven't escalated quite that far...yet. The Atari disk is a different animal from the Apple disk. An Apple copy program, for instance, is an incredibly complex machine language construction, as the Apple CPU controls the disk directly.

The Atari is different; a copy program is very simple. And some companies are using the fact that most people don't know how easy it is to copy an Atari disk to sell copy programs for fairly outrageous amounts. After all, to a user experienced with Apples, the price seems fair.

Frankly, they are a complete ripoff for the price. I do not feel that forty-odd dollars—the cost of a typical Atari disk-cloner program—is a fair price for twenty-odd lines of Basic code.

So much for the copy program makers. Now how about the poor software manufacturer? What do they do, now that I've revealed How To Rip Off Atari Disks? Send me letter bombs?

For the manufacturer's sake, I'll present a few of the latest techniques in copy protection—which the information presented here will now allow someone to copy. And there is no escalation possible; i.e., you won't be able to figure out a way to copy disks protected with this scheme. There is no way, using Atari hardware, to write a program to copy these disks—period. In this way, the word on how to make uncopyable disks will be spread, as will the word on how overpriced (for what they are) the Atari disk copy programs are.

Yes, there are some old copy protection schemes, which we'll discuss, that this information will allow someone to get around. These schemes don't work against the average pirate's copy programs and have not for some time; the disk information discussed here is old news to your average copy enthusiast.

I seriously doubt that anyone will be hurt by this information; if you are going to copy protect a disk, then you might as well use the techniques I will present which make it truly uncopyable, rather than the older ones which the copy programs can duplicate with no difficulty.

Enough philosophy. Let's learn about the disk.

The Disk

Take a disk and look at it. Inside that envelope (sleeve) there is a circular piece of very thin plastic. On that plastic is the same material of which cassette tapes are made: various magnetic substances.

There are 40 "tracks" on the disk. Think of the circular grooves in a record and you'll get the idea of what a track is. There are 40 concentric circles on the disk and each of these tracks is divided into 18 "sectors." The division is done by pie-slicing the track into 18 contiguous segments. So we have 40 tracks with 18 sectors per track, or 720 sectors.

Atari disks are set up to hold 128 bytes of data (a byte, for you beginners, is one character) per sector. In other words, there are 720 x 128 bytes of data on an Atari disk, or 92,160 bytes.

To access a given sector of 128 bytes, that sector must be in the visible portion of the disk (the window cut into the disk sleeve), and the read-write head must move in or out until it's on the right track number. It then reads the sector by examining magnetic fields written to the disk surface.

Please Note that the actual recording surface is the back, not the front, side of the disk. Most people set disks down with the label up, which means the surface that the data is recorded on is being rubbed in the dust and dirt below. Also be very careful not to touch the backside of the disk in the exposed portion.

The Atari talks to the disk over the "serial I/O bus cable." This is the cable you daisy chain from device to device. Now here's the secret of the Atari disk: The disk drive is intelligent. It contains a 6507 microprocessor, a little brother to the 6502 in the Atari itself, that is still capable of quite a lot. It also contains 256 bytes of RAM (read-write memory), and 2048 bytes of ROM (read-only-memory). In short, the disk drive has a complete computer of its own. (For you hardware buffs, there are 128 on the PIA-like chip). How is this different, and why is this good?

On the Apple, the disk drive is "dumb." The main computer must tell the disk head to move to here, the disk to spin to there, and tell the head what information to write. This neatly ties the computer up while the disk is running.

Apple has a standard scheme for storing data, a "standard format," and if you use the Apple routines to read/write data, you'll stay in that format. Alas, that format allows your disk to be copyable, so people have modified the Apple disk routines to make custom weird formats, that cannot be read by copy programs. (And then other folks have written programs, such as Locksmith, to read "uncopyable disks" and then other weirder formats were developed, etc. It's a little like the arms race.)

On the Atari, the two computers work with each other. There are a net total of five, count them, five commands that pass between them. Since the 6507 in the disk drive is helping do the work, the Atari's main processor doesn't have to fiddle around with controlling the disk, and thus can be doing something else at the same time the disk is running. There are no other commands and the disk drive 6507 will ignore any but the five we'll discuss.

Best of all, since the Atari disk controller is "off limits" and may not be programmed, as its program is in read-only memory inside the disk and is not modifiable, you cannot develop strange disk formats. There is one, and only one, Atari disk format. Hence we'll never get into one of these Apple disk escalations, for we cannot control the disk that exactly. Believe me, after viewing the present Apple disk copy mess, we're not missing much.

An ordinary Atari user accesses the disk through what is known as the File Management System or FMS. The user never sees the lowest level of disk commands (the five mentioned) because FMS handles all that for him. Included in FMS is the ability to split the disk sectors, each 128 bytes long, into files, access the files through the operating system, get disk directories of file names, NOTE, POINT, and so on. FMS works in turn through the Five Commands (perhaps a movie should be made by that title?) with the disk drive 6507 to get your disk processing done.

FMS on disk is called DOS.SYS.

What are the Five Commands?

1. Get Status. This returns the status of the disk and in particular of the IN 1771 floppy disk controller chip inside the disk. "What?," you say. Me too. I don't use, and have never used, Get Status. One day I may find out what it's for.

2. Format Disk. This command instructs the disk drive 6507 to lay out the tracks and sectors on the disk, and to "clean off" old data. Think of it as the 6507 laying down record grooves that it can follow later. What actually occurs is that the 6507 writes data across the disk, reads it and makes sure it has stayed the same (this verifies that the disk surface is good), and then writes the sector numbers onto the various sectors of the disk. This information it used later to find a given sector.

3. Get Sector. This command instructs the 6507 to get the entire 128 bytes of a sector and ship them to the Atari through the serial I/O bus cable. The Atari issues the Get Sector command and tells the 6507 what sector number, from 1-720, to fetch. The 6507 then gets busy, spins the disk and positions the read/write head, reads the data, and returns it to the Atari.

4. Put Sector. This command instructs the 6507 to take the 128 bytes about to be shipped down from the Atari and put them onto the sector number specified. The 128 bytes are then sent, and the 6507 positions the disk, etc., and writes the data onto it. You probably do not use this command, you use its relative, which is:

5. Put Sector With Verify. Remember, disks are not all that reliable. So when we put a sector onto the disk with this command, it is first written, then immediately re-read and compared with the original 128 bytes. If they match, fine; all is well. If not, then they didn't record correctly, and an ERROR message results. (In truth, I have never seen this particular ERROR message.)

The Atari's designers decided to add a little audio to this whole process, so whenever a Get Sector occurs (like during a program LOAD), a pleasant "beep" is heard on the TV's audio channel. Whenever a Put sector occurs (during a SAVE, for instance), a not-so-pleasant "clunk" is heard. You'll note that the Put commands always seem to take twice as long as the Get commands; this is because the disk is physically putting, then reading, the information onto the disk for a verify. Hence, it takes twice as long.

In the net time I've worked with the Atari, I've used Put Sector with no verify once, when I was running OSS's operating system. They apparently use this command, and it sounds a great deal different than the standard slow "clunk..clunk.." of the Atari save.

Now *everything* that happens to the Atari disk occurs through these commands.

Atari Diskfile Tutorial — Part III

Disk I/O is a very slow process for the Atari. The disk must be physically spun, which takes a while, and data must be shipped back and forth, which is even slower. The idea behind Atari's DOS is to minimize disk I/O. So we come to the concept of a "buffer."

In Atari's DOS, whenever you read or write to a file, you are reading/writing into a 128-byte reserved area in memory called a buffer. You are not writing to the disk at all. The Atari keeps the contents of one of the sectors of the disk in that buffer. So if you read/write to that buffer, the operation occurs at very high speed, which is what we want. For instance:

```
10 OPEN #1,4,0, "D1:FILE"
20 FOR A=1 TO 10000
30 GET #1,A
40 NEXT A
```

merely reads, byte by byte, 10000 bytes

from disk. But if you run this program, you will note that the disk isn't being accessed continually; only every once in a while will you hear a beep to indicate another read. What happens is that the Atari opens the file and pulls the first sector full of data into the memory buffer. When you do the first 125 GETs, the bytes are pulled out of that memory buffer. Then you try to read another byte, but the Atari doesn't have that one in memory yet. So it requests the disk to send it the next 128 bytes, fills the buffer with those 128 bytes, and starts reading at the beginning of the buffer again.

(If you are wondering why I said the 125th byte, it is because the Atari DOS reserves three bytes per sector for its own uses, which we will talk about later.)

Similarly, when you PUT # to a disk file, the Atari lets you PUT 125 bytes to the buffer, then dumps it to the disk, moves in another 125, and so on.

The result of all this confusion is that the Atari doesn't have to go to disk for every individual byte; rather, it stays in memory for a large number of "disk" accesses, and things run much faster.

When the Atari is LOADing or SAVEing a program, again it uses these buffers. The process is invisible to you, but you can hear the beep as each individual sector in the program is LOADed, or a clunk as a sector is SAVEd. Remember, the Atari can only talk to the disk in terms of complete 128-byte sectors.

Understanding the buffer is important in solving some of the mysterious problems that can occur while using the disk. For instance; let's say you write something out to the buffer, and then your program bombs. You examine the disk file, and find the data never reached the disk. This is because the buffer was never written to disk. If you didn't know about the buffer, you'd be wondering why the disk didn't record what you wrote to it.

The CLOSE Statement

The CLOSE statement is provided for this case. It makes sure that everything in the buffer gets to the disk. It also updates the directory, where all file names are listed.

The DOS program handling all this keeps a table of free sectors on the disk, by the way, and when the buffer fills up and a place is needed to store the 128 bytes, a sector number is taken from that table. When a file is deleted, its sectors are returned to that table. (The XXX FREE SECTORS message at the end of the DOS directory is determined by this).

So much for Basic I/O. Everything done with Basic I/O is a version of the above; everything goes through the 128-byte buffer.

Table 1.

0300 DDEVIC	—Serial Bus ID. Not used by user.
0301 DUNIT	—Disk number. 1-4.
0302 DCOMND	—Command Byte. This is: $21 Format Disk $50 Put Sector, without verify $52 Get Sector $53 Get Status $57 Put Sector, with verify
0303 DSTATS	—Disk Status. This is returned to you after the operation is done.
0304,0305	DBUFLO,HI. Buffer Address. This is a 16-bit address in memory that is the starting address of where to get or put 128 bytes.
0306	DTIMLO Disk Timeout Value. Leave it alone.
0308,0309	DBYTLO,HI Set by handler. Leave it alone.
030A,030B	DAUX1,DAUX2 Sector Number. Which disk sector, 1-720, to read/write.

Program 1.

```
5 REM SAMPLE PROGRAM TO DO DIRECT
6 REM DISK I/O. DAVID SMALL,12/21/81
7 REM
10 DCB=768:REM START OF DISK DC3
20 POKE DCB+1,1:REM DISK 1
30 POKE DCB+2,82:REM $52 = GET SECTOR
35 REM *** GET 128 BYTE OPEN BUFFER
40 DIM BUFFER$(128)
45 DIM CALL$(10)
50 FOR X=1 TO 128:BUFFER$(X)=" ":NEXT X
60 ADDR=ADR(BUFFER$)
70 ADDRHI=INT(ADDR/256)
80 ADDRLO=ADDR-(ADDRHI*256)
90 POKE DCB+4,ADDRLO:REM BUFFER ADR LOW
100 POKE DCB+5,ADDRHI:REM BUFFER ADR HI
110 REM *** SECTOR NUMBER ***
120 PRINT "INPUT SECTOR NUMBER"
130 INPUT SECTOR
140 SECTORHI=INT(SECTOR/256)
150 SECTORLO=SECTOR-(SECTORHI*256)
160 POKE DCB+10,SECTORLO
170 POKE DCB+11,SECTORHI
200 REM *** BUILD SHORT ASSY PROGRAM
210 REM *** OF  PLA, JMP DSKINV.
230 CALL$(1)=CHR$(104):REM PLA
240 CALL$(2)=CHR$(32):REM JSR
250 CALL$(3)=CHR$(83):REM $53
260 CALL$(4)=CHR$(228):REM $E4
270 CALL$(5)=CHR$(96):REM RTS
300 X=USR(ADR(CALL$))
310 DSTATS=PEEK(DCB+3)
320 PRINT "DISK STATUS=";DSTATS;" (1=OK)"
330 PRINT "DATA:"
340 PRINT BUFFER$
341 REM *** CLEAN OUT BUFFER$
342 FOR X=1 TO 128:BUFFER$(X)=" ":NEXT X
350 GOTO 110
400 END
```

How about DOS?

When you issue the COPY disk command from DOS, each sector of the file is sent to memory. In other words, DOS copies the contents of that file into memory, 128 bytes at a time. It then takes the file in memory and writes it out to disk, from memory, again in 128 byte chunks.

When you issue a Duplicate Disk command, DOS reads in every sector on the disk that is marked as "used" and stores it in memory. The unused sectors are ignored

Table 2.

Byte 1: Flag byte.

Bit 7 = 1 if this file has been deleted
Bit 6 = 1 if this file is in use
Bit 5 = 1 if this file is locked
Bit 0 = 1 if this file is OPEN.

Bytes 2,3. Sector Count; the number of sectors in the file, stored low, then high bytes.
Bytes 4,5. Starting Sector. Where the file begins (what sector number).
Bytes 6-16. File name. Last three bytes are the extension if you add one, otherwise they are blanks.

Table 3.

Byte 1 .. Data	
Byte 125 ..	
Byte 126	File Number (6 bits) and high two bits of "forward pointer"
Byte 127	Forward Pointer
Byte 128	Short Sector count .. indicates this sector not completely full.

to save time. DOS then writes those used sectors out to the new disk, in the same position it found them on the old one, and doesn't worry about the empty sectors. Since there are 92,000 bytes on a disk, and only 48K maximum of memory, it may have to do this in smaller pieces—let's say 32K at a time. You then must physically change the disk several times on a single-drive system. If you are using a multiple disk system, you will see it do the copy in multiple stages.

Note: If you are copying disks for backups, remove any cartridge; each cartridge selects 8K of RAM and makes for more disk swaps.

In order to know what sectors are used, and what files are on the disk, some tables must be kept on that disk. The DOS reserves certain sectors on the disk for these tables. One is known as the VTOC (Volume Table of Contents) and is sector number 360. In it is a table of all 720 sectors on disk, stored as bit-map; i.e., one bit per sector. The disk directory is stored in sectors 361-368; this is the actual list of the file names and the sectors they occupy. To find a file, DOS must use this directory.

By the way, these reserved sectors explain why only 707 sectors are available on a "clean" disk; the rest are used for the directory tables, etc. You will also hear, during the disk formatting process, right near the end, the clunks as DOS writes out a fresh directory to the disk, after the disk 6507 has finished formatting it.

Remember, DOS at the lowest level, past all of the directory opens, XIO's, and so forth, is only doing get sectors/put sectors. Let's learn about those calls.

Get Sector/Put Sector Calls

All get sector/put sector calls rely on a table called the Device Control Block. Data is put into this table, and a jump is made into the operating system, which in turn uses that table. Table 1 is the disk table, starting at $300 (or 768 decimal).

One POKEs values into this table, and then does a JSR to DSKINV ($E453). DSKINV then does the requested operation and returns to you. Program 1 is a sample program that does the needed POKEs and requests from you a sector number (1-720) to read.

This program POKEs the DCB parameters, reserves 128 bytes in memory as a string for the buffer area, requests the sector number, then runs a very short assembly program which does a PLA, needed by Basic, then jumps off to DSKINV. DSKINV then does the operation, and returns to Basic.

There the status code placed in the table by DSKINV is printed, and the buffer area, as a string, is also printed. Sure, it will be strange, but you can examine individual bytes easily. If you read one of the directory sectors (try 361) you will see the directory entries.

If an 87 instead of an 82 were POKEd into DCOMND, then a Put Sector would occur ($57=Put Sector, $52=Get Sector). Then the contents of that buffer would be written to disk.

The status code will be a 1 if all goes well. If there is a problem, you may find a 144 as the error code. This Device Not Done Error occurs if the disk is bad, the disk is write protected, etc.

We have just done disk access the same way DOS does it, at the sector level. Note we have not done an OPEN or an XIO; we have directly accessed the disk. This can be handy if you have a need for a disk allocation scheme free of DOS; I use it to store directly fixed-length records. It is fast and efficient, and removes the DOS overhead.

Also, it enables you to access the directory and modify it as necessary; this allows you, for instance, to un-delete files.

Because we are not using DOS, but are still accessing the disk, we are not dependent on the DOS directory or "In-Use-Table." We can read or write any sector on the disk, regardless of what DOS thinks that sector is for. You may want to take note: This is how to write a "boot record," a special disk record that enables your disk to boot by itself, as so many games do. The boot record is record 1; see the hardware manual for further details.

Now that you can access the directory data directly, you can see all the data stored rather than just what OPENing "*.*" returns. A directory entry is 16 bytes long for each file. There are eight entries per sector in the directory, and eight directory sectors, hence 64 files maximum. See Table 2.

As an example of how the directory is used, if a file is LOCKed from DOS, all that occurs is that its flag byte has bit 5 set to 1.

When you delete a file, all that occurs is that the delete flag on the file is set. Then, next time DOS needs a place to put a new file entry, it knows it can overwrite the current entry.

Reading the Data into a File

In order to read the data in this file, one goes to the Starting Sector number found in the directory. Each sector contains information as shown in Table 3.

The "forward pointer" is where the next sector of the file can be found. It is a sector number from 1 to 720. When that next sector is read in, it in turn tells where the next sector can be found, and so on. This is called a "linked list" or chained sector scheme. This way, files don't have to be in any particular order on the disk (e.g., running continuously from sector 30 to 40). The sectors can jump around all over the disk, yet to DOS they are still linked together.

The file number is the file number in the directory. It is set to the same number as the entry number in the directory for all sectors. This is a safety measure. Let's say we are reading directory entry 6, and are going along, sector by sector. In the file number position in a given sector, we suddenly find a 13 (or whatever) instead of the 6 that should be there.

We know that the data for this sector has gotten scrambled, and the sector chain terminates there. This is an ERROR 164, which I am sure you have seen before; it is a warning that the data in this sector are likely to be bad. The sector link is also likely to be bad, so DOS normally stops an ERROR 164.

Reading in sectors one at a time, by getting the next sector number, reading, and so on, is called "sector tracing." Atari DOS does it every time it reads in a file. Atari also sells a "disk fix" program through their program exchange which uses the above information; you now know enough to use it. For instance, the forward "link" of a given section may be modified if desired.

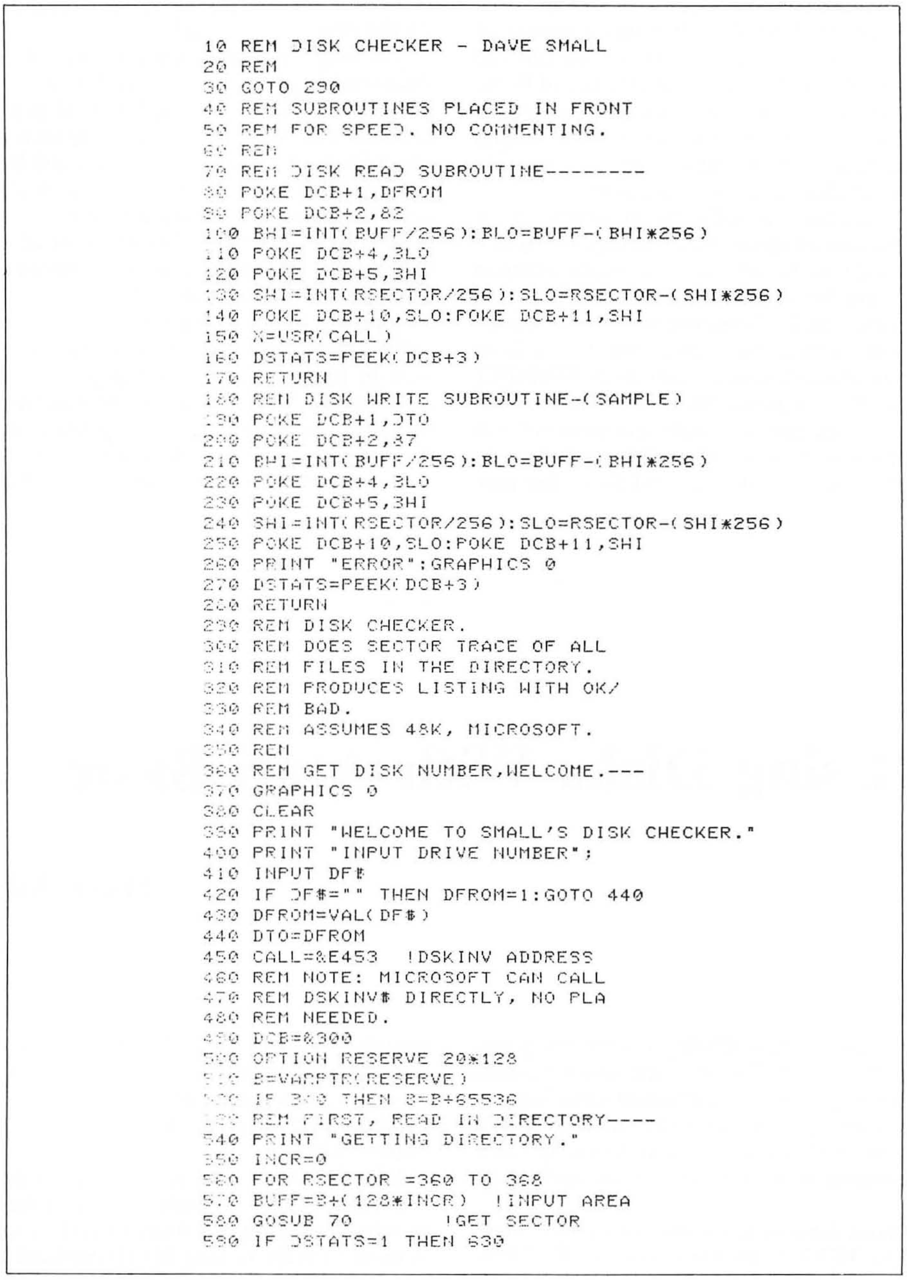
We can do the same sector tracing out of Basic. We will find a given directory entry, read it in starting at the sector number given by the directory and proceed through each sector, with each new sector number obtained from the data in the previous one, until we have read in the number of sectors specified in the directory. Should anything be wrong (file number, etc.) we will know the file has gone bad.

Program 2 illustrates first how to read in the directory and interpret it, and second how to follow a sector chain for a given file. It reads in the directory, lists it to the screen, and asks which file to trace; if A is entered, all files will be traced (this takes a while). It will then trace the file through, stopping at any "broken links" or bad file numbers. It is an excellent way to check a disk which has some files going bad, and find out which ones are still readable and which aren't. After each check the program produces a directory listing showing which files are good, bad, or still unknown.

The techniques used within the program will probably prove more useful than the program; however, remember that this is an example. It can serve as the basis for custom disk inspect/modify routines if you wish to write them. Also, the program is written in MicroSoft Basic but translating it to the old Basic should be no problem at all. I highly recommend MicroSoft Basic in terms of speed and features as compared to the old Basic; an in-depth review of the Atari version of MicroSoft Basic will appear in an upcoming issue.

Sadly, the software pirate community also found out about direct disk I/O, literally years ago. Back then manufacturers were copy protecting disks by fouling up the directory and sector maps so that DOS

Program 2.



```
1Ø REM DISK CHECKER - DAVE SMALL
2Ø REM
3Ø GOTO 29Ø
4Ø REM SUBROUTINES PLACED IN FRONT
5Ø REM FOR SPEED. NO COMMENTING.
6Ø REM
7Ø REM DISK READ SUBROUTINE--------
8Ø POKE DCB+1,DFROM
9Ø POKE DCB+2,82
1ØØ BHI=INT(BUFF/256):BLO=BUFF-(BHI*256)
11Ø POKE DCB+4,BLO
12Ø POKE DCB+5,BHI
13Ø SHI=INT(RSECTOR/256):SLO=RSECTOR-(SHI*256)
14Ø POKE DCB+1Ø,SLO:POKE DCB+11,SHI
15Ø X=USR(CALL)
16Ø DSTATS=PEEK(DCB+3)
17Ø RETURN
18Ø REM DISK WRITE SUBROUTINE-(SAMPLE)
19Ø POKE DCB+1,DTO
2ØØ POKE DCB+2,87
21Ø BHI=INT(BUFF/256):BLO=BUFF-(BHI*256)
22Ø POKE DCB+4,BLO
23Ø POKE DCB+5,BHI
24Ø SHI=INT(RSECTOR/256):SLO=RSECTOR-(SHI*256)
25Ø POKE DCB+1Ø,SLO:POKE DCB+11,SHI
26Ø PRINT "ERROR":GRAPHICS Ø
27Ø DSTATS=PEEK(DCB+3)
28Ø RETURN
29Ø REM DISK CHECKER.
3ØØ REM DOES SECTOR TRACE OF ALL
31Ø REM FILES IN THE DIRECTORY.
32Ø REM PRODUCES LISTING WITH OK/
33Ø REM BAD.
34Ø REM ASSUMES 48K, MICROSOFT.
35Ø REM
36Ø REM GET DISK NUMBER,WELCOME.---
37Ø GRAPHICS 0
38Ø CLEAR
39Ø PRINT "WELCOME TO SMALL'S DISK CHECKER."
4ØØ PRINT "INPUT DRIVE NUMBER";
41Ø INPUT DF$
42Ø IF DF$="" THEN DFROM=1:GOTO 44Ø
43Ø DFROM=VAL(DF$)
44Ø DTO=DFROM
45Ø CALL=&E453   !DSKINV ADDRESS
46Ø REM NOTE: MICROSOFT CAN CALL
47Ø REM DSKINV$ DIRECTLY, NO PLA
48Ø REM NEEDED.
49Ø DCB=&3ØØ
5ØØ OPTION RESERVE 2Ø*128
51Ø B=VARPTR(RESERVE)
52Ø IF B<Ø THEN B=B+65536
53Ø REM FIRST, READ IN DIRECTORY----
54Ø PRINT "GETTING DIRECTORY."
55Ø INCR=0
56Ø FOR RSECTOR =36Ø TO 368
57Ø BUFF=B+(128*INCR)  !INPUT AREA
58Ø GOSUB 70           !GET SECTOR
59Ø IF DSTATS=1 THEN 63Ø
```

couldn't perform the copy. With the advent of direct disk I/O, all these schemes were bypassed with what is known as a "sector copier."

As a side note, a sector copier is sometimes advertised as a "nibble copier" or "byte copier." This is an Apple term, and refers to individual bytes being read off the disk. It is completely inaccurate when applied to the Atari; all Atari disk I/O is in 128-byte blocks.

A sector copier, as you have guessed, just issues a read of all the possible sectors, then writes them.

Sector Copying

The program is simple to write now that you understand about sector I/O and how to call up DSKINV. It is just a matter of reading in 720 sectors and writing out 720 sectors. As I said, about 20 lines of Basic code are required. And now you understand why, as I mentioned last month, selling such a program, at the prices currently being charged, is such a ripoff.

But...how can a disk be protected against this sort of thing? Fortunately, there is a good way. Several software manufacturers hit on the same idea at the same time: bad sectoring. Let's assume we have a special disk, with some sectors on it that have never been formatted, as with the FORMAT DISK command. Most of them are OK; just a few are bad, as if there were holes in the disk. Now when the disk is told to read these sectors, it can't find them, for they were never formatted. So it returns an ERROR 144, instead of a normal return (1). Then, in the software on the disk, that sector is called, and if an ERROR 144 doesn't result, the program quits.

Let's say the program is then copied to another disk by an average sector copier. The "bad" spots on the disks will not copy; they will return ERROR 144, but on the destination disk, there will still be formatting information on those sectors. This is because the average Atari user cannot create a disk with bad spots in it; the FORMAT command is handled completely inside the disk by the disk 6507 microprocessor. The program will find that where it expected a bad sector, it now finds a good sector. Hence the program knows it is not residing on the original disk, and can quit.

The only way to defeat this scheme is to disassemble code and remove the sector check, and even if the author is slightly clever, it will take so long to find the sector checking routine that the program will be outdated by the time it is finally made copyable. Besides, consider what a person's time is worth compared to the cost of a program; if it takes two weeks to break a copy protection scheme, isn't it a better idea just to buy the program?

This bad-sectoring scheme works quite well in preventing a sector copier from being useful. Sure, a copy of the disk may be made, but without the bad spots on the disk, it will never run. And your average user cannot create these bad spots, for only access to the disk controller program allows that, and only very advanced users can accomplish this.

Atari currently uses this bad sector scheme in several of their disk-based programs. *Jawbreakers* uses the same scheme (hence the disk retry, or "snaaark," sound when you first boot it up). *The Wizard and The Princess* use it also. In other words, the move is toward this sort of protection.

Software manufacturers can probably figure out a way to write some bad sectors onto a disk. For instance, what popular personal computer gives you complete control over the disk read/write process? There are other ways, too.

Another manufacturer doesn't bother copy protecting his disks at all. However, a small module must be plugged into the front joystick port before the program will run. The module probably contains some sort of ROM-type circuit which is then read by the program through the PIA chip (joystick ports).

Well, we have covered a great deal, from piracy philosophy to sector chaining in these two columns. I hope you have enjoyed it and have learned something about the Atari disk, and how it really works. I also hope I have described some effective protection schemes for software manufacturers.

Direct disk I/O is a very powerful tool for Atari users; use it to make your programs faster and more efficient. □

Using Disks With Atari Basic

Dave Johnson and Embee Humphrey

This article will help you write programs that use data files that are saved on disk. We hope you will use this information as a beginning point for experimentation. We believe that the only way to really learn programming is to try things and make mistakes in the process. We have included answers to what we have found to be the most common questions.

These are the Atari disk input/output commands:

OPEN #reference number, open code, 0, filespec: The first parameter, #reference number, is a number between 1 and 7 that is used to refer to that file throughout a program. It must be preceded by the "#" symbol. The second parameter, open code, is a number that tells the computer if you want to read, write, etc. See Table 1.

The third parameter is reserved for special control codes that will not affect you in disk operations. Always use a 0 for this third parameter. The last parameter, filespec, is the device and file name. Filespec

David Johnson and Embee Humphrey, Atari, Inc., 1272 Borregas Ave., Sunnyvale, CA 94086.

can be a string variable, or the device and filename enclosed in quotes. Example: You may declare A$= "D:FILENAME" and use A$ in the OPEN statement, or you may just use "D:FILENAME" itself.

CLOSE #reference number: This tells the computer that #reference number is no longer being used. (Note—if you are writing data, and fail to close a file, any data in the buffer will not be saved on disk.)

PRINT #reference number;[What you want printed]: This is used the same way as a regular Basic PRINT statement. The #reference number tells the computer to PRINT to the file you have opened using that reference number.

INPUT #reference number; [Input list]: This works like a regular Basic INPUT statement. The #reference number tells the computer to look for input from the file opened using that reference number.

NOTE #reference number, variable for SECTOR location, variable for BYTE location.

POINT #reference number, SECTOR variable, BYTE variable: NOTE and POINT are explained later.

OPENing a file tells the computer that you are going to use a particular file in a particular way, and that you will refer to that file in your program using a particular reference number. (Atari Basic manuals call this reference number an IOC3 number, for Input/Output Control Block.)

Table 1.

Open Code	Function
4	Input. (Read from the file only).
8	Output. (Write to the file only). NOTE - this erases any data in this file. To OPEN a file to add data, use 9 for APPEND.
12	Both input and output. (Read and write).
9	Append - add to end of file. (Write, beginning at end of file).

For example, if you want to read from file TEST.DAT, and you decide to refer to that file as number 4, you would use the following open statement example. (The number doesn't matter as long as whenever you want a certain file, you refer to it with the same number.)

An example of OPEN: OPEN #4,4,0, "D:TEST.DAT". This means OPEN a file, read data from the file, the file will be on device "D;" (disk), whose name is TEST. DAT, and remember that this file will be referred to as #4 for future I/O operations.

To read from the file you would say: INPUT #4; expressions....

And finally, when you are done with the file: CLOSE #4.

INPUT and PRINT

INPUTing and PRINTing with the disk is the same as inputting from the keyboard and printing to the screen. The #reference number tells the computer that you are using the disk instead. (You must have OPENed a disk file with that number).

The easiest way to experiment with this is in DIRECT mode, typing in statements without line numbers. Now try a few things:

OPEN #1,8,0,"D1:TEST.DAT": This opens a file, TEST.DAT, for write only. It will be referred to later as #1.

Figure 1.

```
DIM S$(20)
OPEN #1,4,0,"D:TEST.DAT"
INPUT #1,S$   Read the first line.
INPUT #1,S$   The second.
INPUT #1,S$   The third.
INPUT #1,S$   And the APPENDED line to verify this.
PRINT S$
```

PRINT #1;"THIS IS A STRING": This writes this message to the file.

PRINT #1; 12345.6: This writes the number 12345.6 into the file.

PRINT #1;"THIS IS ANOTHER STRING";12345.6: This writes both the string and the number to the file with the same statement.

CLOSE #1: This closes the file.

To see what is on the disk: OPEN #2,4,0, "D:TEST.DAT".

We used #2 here to demonstrate again that the number itself does not matter. What matters is that the same number be used for all references to *that* file after the OPEN.

Since you will be reading strings, you need to DIM a string to read into: DIM S$(30)

Now, to read it, type: INPUT #2,S$. Then type: PRINT S$, and the screen displays: THIS IS A STRING, which is what you put on the disk.

Now, a surprise! We told you that PRINTing to disk was the same as printing to the screen. When you type PRINT #4; 12345.6, you put the CHARACTERS "12345.6" on the disk, not a binary representation.

Now type:

INPUT #2,S$
PRINT S$

and the screen displays: 12345.6

Reading the number into a string gets the string representation of the number that was on the disk. Reading the number into a number variable would also have worked. If the binary representation of the number was put on the disk, you would not have been able to read it into a string and see it. This is important to remember, because putting the number 10 on the disk takes two bytes of disk space, while putting the number 123 on the disk takes three bytes. This will be important in the discussion of random access.

Remember: the number of digits is equal to the number of bytes the number will take up on the disk.

The next example shows another way that similarity between screen and disk output may surprise you. From the above example the file is still open:

Type:
INPUT #2;S$
PRINT S$

and the screen displays: THIS IS A STRING 12345.6

It read into the one string *both* the string and the number that you had put on the disk. When you put this data into your file using PRINT#1;"THIS IS A STRING";12 345.6, it went onto the disk exactly as it would have appeared on the screen. When reading it, the computer had no way of knowing where the string stopped and the number started on the disk.

You could not have said INPUT #2;S$, NUM because inputting S$ would have taken you past the number so, if you are going to read data back as different lines, put it on the disk as different lines.

APPEND

Now you can experiment with APPEND. You already have a file, TEST.DAT, on the disk. If you want to add data to this file you must use APPEND, because opening the file to write again (with an 8) will erase what you have already put into it.

In OPEN #5,9,0,"D:TEST.DAT", the 9 means APPEND.

Now, try this. Type:

PRINT #5;"THIS IS ADDED"

CLOSE #5

To see what happened, type the lines in Figure 1. The screen displays: THIS IS ADDED

Trying to type INPUT #1,S$ one more time would cause an ERROR 136, which means END OF FILE.

CLOSE #1

NOTE and POINT

NOTE #reference number, sector variable, byte variable: This takes NOTE of where the disk read/write head is physically positioned. The #reference number is the IOCB number. If you want to mark where you are in a file opened as #1, you say NOTE #1,variable1,variable2. Variable1 saves the sector and variable2 saves the byte.

POINT #reference number, sector variable, byte variable: This physically POINTs the disk head to the sector and byte. If that sector and byte are not located within the file opened with that #reference number, you get an error message.

Figure 2.

```
DIM S$(20)
OPEN #1,4,0,"D:TEST.DAT"
NOTE #1,X,Y      take note of where file starts.
INPUT #1,S$
PRINT S$
The screen displays: THIS IS A STRING
```

NOTE and POINT are useful for repositioning the read/write head to the beginning of a file. For example, if you want to read the first string in your TEST.DAT file twice before proceeding, you would type the lines in Figure 2.

The screen displays: THIS IS A STRING

If you read again, you will be reading the second item in the file, which is: 12345.67.

To point back to the beginning:

POINT #1,X,Y

INPUT #1,S$

PRINT S$

and the screen will again display: THIS IS A STRING, which is the first record in the file!

Random Access

NOTE and POINT are useful for random access. If you keep track of where you put data, using NOTEs, you can later get at the data, using a POINT, without reading everything in front of it.

Briefly, suppose you have a mailing list. Each time you save a record, you NOTE where it is physically written, and save those sector and byte numbers in arrays—SECTOR(record number),BYTE(record number). Then you save those arrays in another file on the disk.

To look up records, you would first read the array data file to put the information back into the arrays. You would then look up the individual records by POINTing directly at them. (SECTOR and BYTE are the variable names we have assigned to our arrays.)

X=SECTOR (indexed by record number). (To read the 23rd record you would use X=SECTOR(23).)

Y=BYTE (indexed by record number)

POINT #reference number,X,Y

and you are ready to read that record.

Be careful. If you want to modify a record and write it back into the file in the same place, you must not change the length of that record. If it is changed, it could overlap into the next record, overwriting the data there. Remember that numbers use as many bytes as there are digits.

Before you modify records, you must format numbers into strings with leading blanks or zeroes that are always the same length. The subroutine in Figure 3 accomplishes this. Before calling, put the number you want formatted into the variable DSKNUM. The number will be formatted into a string of length 10 with leading zeroes and placed in string variable DSK$.

Figure 3.

```
1000 L = LEN (STR$ (DSKNUM )) : REM GET THE LENGTH
1010 DSK$ (1,10-L) = "0000000000" :REM PUT ZEROES IN FRONT OF
       STRING
1020 DSK$ (11-L) = STR$ (DSKNUM) : REM PUT NUMBER RIGHT JUSTIFIED
       INTO STRING
1030 RETURN
```

Traps

Atari Basic has a TRAP command. The format for the TRAP command is: TRAP line number.

To TRAP to a line number means to go to that line number if an error occurs. This can be useful for disk operations. For example, if you want to go to a certain line number when you reach an END OF FILE, you can take advantage of the ERROR 136 that happens when you try to read past the end. See Figure 4.

Figure 4.

```
10 DIM S$(20)
20 OPEN #1,4,0,"D:TEST.DAT#:REM OPEN THE FILE
30 TRAP 70:REM ON END OF FILE ERROR, GOTO 70
40 INPUT #1,S$:REM READ FROM THE FILE
50 PRINT S$:REM PRINT TO THE SCREEN
60 GOTO 40:REM KEEP DOING THIS
70 CLOSE #1:REM END OF FILE REACHED, CLOSE FILE
```

Part V
User Programs

Mazemaster: Maze Making and Running

Fred Brunyate

Shortly after buying an Atari 800 computer, I bought *Basic Computer Games* by David H. Ahl (published by *Creative Computing Press*). After taking care of a few essentials like *Life* and *Hammurabi*, I decided to adapt the *Amazin'* program so that I could use the joysticks to run the maze.

My first study of the program failed to provide any clear idea of the program's logic. The listing lacks even the most rudimentary documentation. Frustrated, I entered the program line for line, changing only the graphics characters. It really did work. Even working, however, the "why" of the program's complicated tree structure and repeated code remained obscure. The project went to the back burner.

Inspiration arrived while I was trying to fall asleep one night. Start with a maze with no paths, all single cells. From the selected starting cell, perform a random walk through the cells, marking each cell you move into and removing the intervening wall. Do not move into a marked cell or off the edge. If you cannot move, select any marked cell and resume the random walk. Continue until all cells are marked. Select any cell as the finish point, and you are done.

As I began laying out my own maze building program, two features of Atari Basic helped the whole routine fall into place. First, Atari Basic arrays have a zero row and a zero column. By adding an extra row and column and setting the elements of all four non-zero, all of the explicit boundary checking disappears, without having to remember any co-ordinate modifiers. Second, Atari Basic allows variable names to be the object of GOSUB statements. The starting line numbers of the move subroutines for all legal moves can be stored in an array. Then, with N a random number, ON N GOSUB JUMP(1),JUMP(2), JUMP(3), makes a classic random walk.

The printing routine works the same as the book version. With the top and left boundaries assumed, it needs information on only two of the four sides: neither, right only, bottom only, or both sides open. These new sides then become the top or left sides of other cells.

The maze running routine used another Atari feature. Neither of the arrays is used to check for legal moves. With the LOCATE statement, the maze displayed on the screen (or, more accurately, the display list of the screen) can be examined directly to detect walls and openings.

Now for the gory details. The main program (lines 1-999) begins with all the things that need to be done only once: DIMensioning arrays, setting the boundaries non-zero, adjusting the margins, setting constants and subroutine names. Three subroutine calls do all the real work. Finally, print the results and repeat on request. Nothing fancy so far.

The MAZEPRINTER routine (lines 1000-1999) uses the SETCOLOR statement to display the maze at the same color and intensity as the background. It is all there, you just cannot see it until the second SETCOLOR statement causes the completed maze to appear suddenly. In earlier versions of this program, you could solve most of the maze before the last line was printed and the timer started. Each cell of the maze is two print positions on a side. The routine prints spaces or graphics characters according to the information stored in array V by the MAZEBUILDER routine. Refer to the program comments for a step-by-step description.

The MAZEBUILDER routine (lines 2000-2999) uses the array W to simulate the cells of the maze. It marks a cell by setting the corresponding element of W non-zero, and removes a wall by changing the corresponding element of array V. After selecting a starting cell in the first row and initializing the cell counter, it checks for a zero neighbor cell in all four directions. For each zero cell found, the appropriate subroutine line number is added to the list kept in the array JUMP. There will be a maximum of three entries; the fourth direction leads to the previous cell. A subroutine is randomly selected from the list if it contains more than one cell. That subroutine is executed to move into the new cell, mark it non-zero, and change V to remove the intervening wall. This continues until no zero neighbors are found.

When the random walk is blocked (no zero neighbors), the routine examines cells at random until a non-zero cell is found and resumes the random walk from this point. This assures us that the new path segment will be connected to the original path. Since the routine will not move into a non-zero cell, each path segment is connected at only one end, and There is exactly one path between any two cells.

The routine halts when 95 percent of the total cells have been added to the path. This leaves some cells unconnected, but saves time by not trying to find those last few cells. The maze exit is a randomly selected non-zero cell in the last row. Actually, any two non-zero cells could be selected as the start and finish.

The MAZERUNNER routine (lines 3000-3999) handles the joystick input, checks for legal moves, displays the trial, and times the run. When decoding the joystick input, I find it easier to work with the complement of the value returned, i.e. "D=15-STICK(0)". The x direction is +1 if GT 7, 0 if D LT 3, otherwise -1. The y direction is +1 if D MOD 2=0, 0 if D MOD 4=0, otherwise -1. The routine uses an inverse video asterisk as a CURSOR with the joystick, and a SPOT, a normal asterisk, is left behind.

If the character, determined by the LOCATE statement, at the new position plus the joystick input, is not a BLANK or a SPOT, the routine ignores the joystick input. You can not move through walls. Otherwise, it plots the CURSOR at the new position and a

Fred Brunyate, 6076 Marsh Rd., Apt. E-2,
Haslett, MI 48840.

SPOT at the old position. It continues monitoring the joystick until the CURSOR co-ordinates match the finish conditions. The real-time clock is read immediately before and immediately after running the maze and the total time is computed. The three bytes are read as quickly as possible to minimize the chance of an error caused by one of the bytes rolling over to zero.

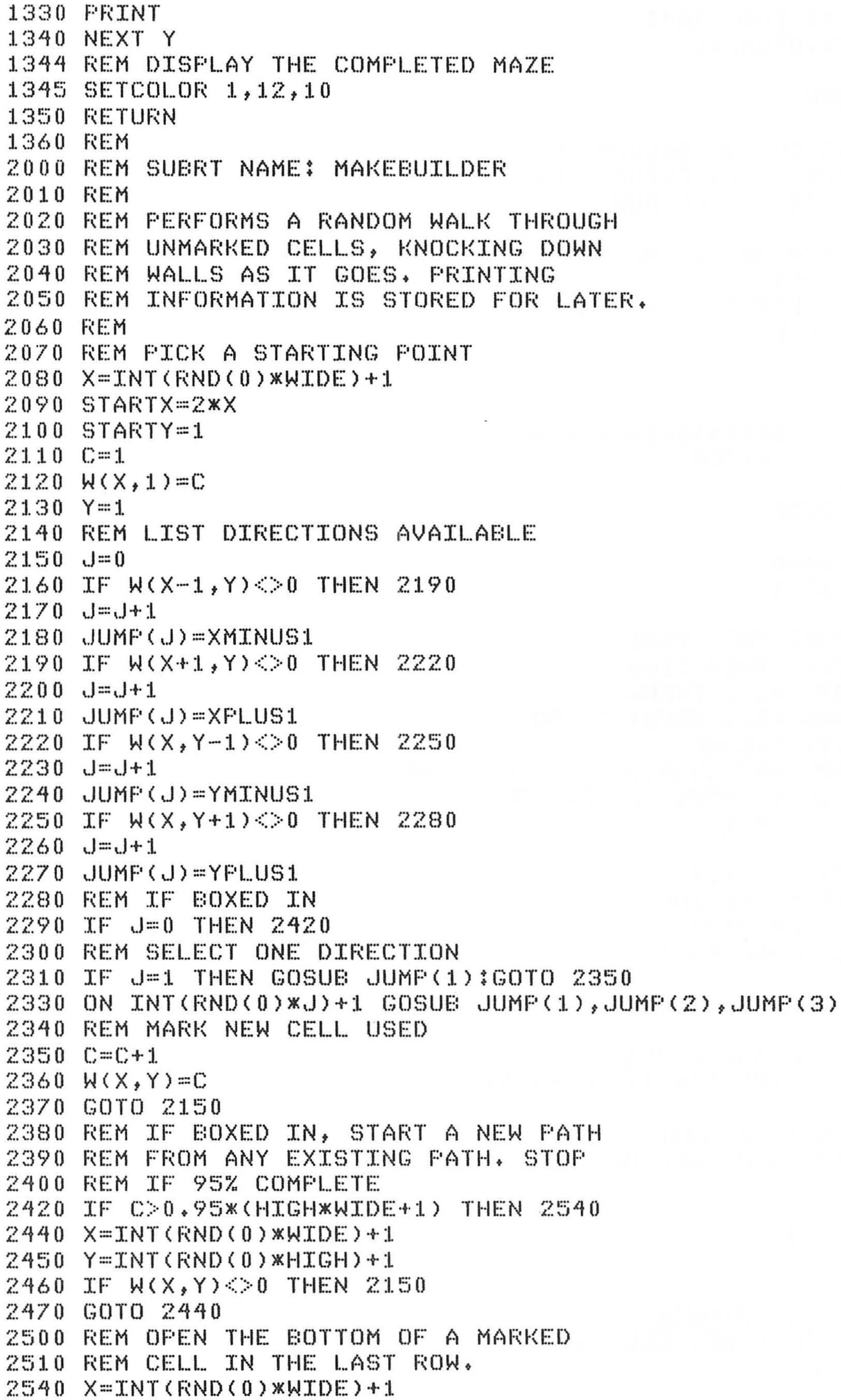
The subroutines at lines 4000-4999 do the moving from cell to cell and take out the walls during the random walk. To move right (XPLUS1) or down (YPLUS1), first change V to open the wall, then change the co-ordinates. To move up (YMINUS1) or left (XMINUS1), first change the co-ordinates, then change V.

While this program is an enjoyable game as it is, it is also a starting point for other programs. The random walk technique is easily expandable to three dimensions (or four, if you can name them). A light pen would be ideal for running the maze. The start and finish points can be moved. And a little imagination will turn the cells into rooms in a dungeon or castle.

Variables	Description
HIGH, WIDE	The maze size, in cells. 17 X 10 fills the screen nicely.
V(WIDE, HIGH)	This array indicates which walls have been removed. The top and left side of each cell is assumed. The bottom and right sides are indicated as follows: 0—neither side open 1—bottom only open 2—right side only open 3—both sides open
W(WIDE+1,HIGH+1)	This array is used to build the maze. Zero elements are available. The outside rows and columns are set non-zero.
C	The number of cells attached to the path.
JUMP(3)	This array contains the list of subroutine line numbers (XMINUS1, YMINUS1, XPLUS1, YPLUS1) from which to select the next move of the random walk.
Y$(1)	The user response, 'Y' or 'N'.
CURSOR, SPOT, BLANK	Characters used by MAZERUNNER to check for and plot the trial.
FIRSTTIME, LASTTIME	The clock readings before and after the run.
TIME	The time of the run, in seconds.
TRIES, TOTALTIME	The number of mazes run and the total time used. Used to figure the average time.
X,Y	The current cell during MAZEBUILDER and MAZEPRINTER. The current screen position during MAZERUNNER.
STARTX, STARTY	The screen co-ordinates of the beginning of the maze.
BOTTOM	The screen row of the last line of the maze.

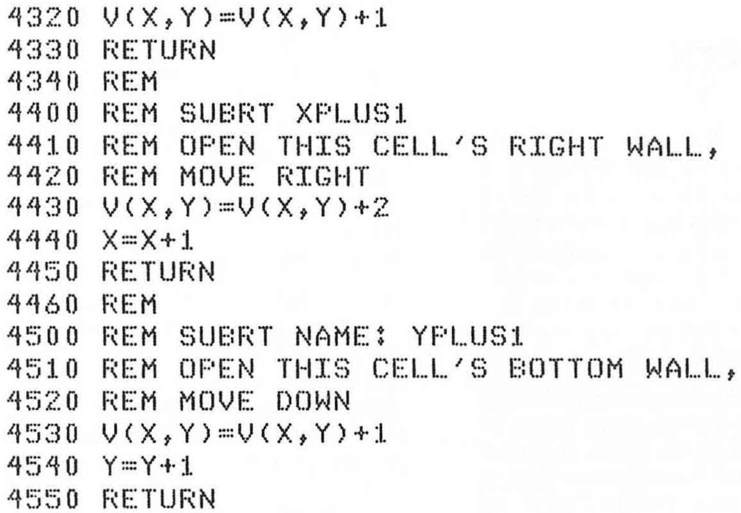
```
10 REM MAZE MASTER
15 REM
20 REM FRED BRUNYATE
30 REM HASLETT, MICHIGAN
40 REM 1981
50 REM
100 REM POKE NEW MARGINS, SET SUBROUTINE LOCATIONS,
    TOTALS, AND CONSTANTS
105 WIDE=17
106 HIGH=10
110 DIM V(WIDE,HIGH),W(WIDE+1,HIGH+1)
120 DIM Y$(1),JUMP(3)
130 POKE 82,1
140 POKE 83,38
150 MAZEPRINTER=1000
160 MAZERUNNER=3000
170 MAZEBUILDER=2000
190 XMINUS1=4200
200 YMINUS1=4300
210 XPLUS1=4400
220 YPLUS1=4500
230 TRIES=0
240 TOTALTIME=0
250 SPOT=ASC("*")
260 CURSOR=ASC("*"):REM THIS IS INVERSE VIDEO
270 BLANK=ASC(" ")
280 BOTTOM=2*HIGH+1
300 REM SET BOUNDARIES NON-ZERO
310 FOR I=1 TO WIDE
320 W(I,0)=-1:W(I,HIGH+1)=-1
330 NEXT I
340 FOR I=1 TO HIGH
350 W(0,I)=-1:W(WIDE+1,I)=-1
360 NEXT I
490 REM
500 REM CLEAR THE ARRAYS AND SET BOUNDARIES
505 GRAPHICS 0
510 PRINT :PRINT "I'M THINKING UP A GOOD ONE."
515 PRINT :PRINT "      DON'T GO AWAY."
520 FOR I=1 TO WIDE
530 FOR J=1 TO HIGH
535 W(I,J)=0:V(I,J)=0
540 NEXT J:NEXT I
590 REM
700 REM
710 GOSUB MAZEBUILDER
720 GOSUB MAZEPRINTER
730 GOSUB MAZERUNNER
740 REM
800 REM DISPLAY RESULTS
810 TRIES=TRIES+1
820 TOTALTIME=TOTALTIME+TIME
830 AVTIME=INT(TOTALTIME/TRIES*100)/100
840 POSITION 1.BOTTOM+1
```

```
850 PRINT "TRY #";TRIES;" TIME WAS: ";TIME;" SECONDS."
860 PRINT "   AVERAGE TIME: ";AVTIME
870 REM
900 REM ASK USER VITAL QUESTIONS
910 PRINT "      REPEAT THIS MAZE (Y OR N)";
920 INPUT Y$
930 IF Y$="Y" THEN 720
940 PRINT "      ANOTHER MAZE (Y OR N)";
950 INPUT Y$
960 IF Y$="Y" THEN GOTO 500
970 END
990 REM
1000 REM MAZEPRINTER
1010 REM
1020 REM VALUES IN 'V' DETERMINE THE
1030 REM RIGHT AND BOTTOM WALLS OF
1040 REM EACH CELL:
1050 REM V(I,J)=0;  NO OPENINGS
1051 REM V(I,J)=1;  BOTTOM OPEN
1053 REM V(I,J)=2;  RIGHT OPEN
1054 REM V(I,J)=3;  BOTH OPEN
1055 REM
1060 REM POKE TURNS OFF THE CURSOR
1061 REM SETCOLOR HIDES MAZE UNTIL
1062 REM PRINTING IS DONE.
1080 GRAPHICS 0
1090 POKE 752,1
1091 SETCOLOR 1,9,4
1095 REM PRINT THE TOP LINE
1100 FOR X=1 TO WIDE
1120 PRINT "";:REM CTRL-S,CTRL-R
1150 NEXT X
1160 PRINT "":REM CTRL-S
1165 REM PRINT THE LEFTMOST WALL, THEN
1166 REM A CELL AND WALL OR OPENING.
1170 FOR Y=1 TO HIGH
1180 PRINT "|";:REM SHIFT-=
1190 FOR X=1 TO WIDE
1200 IF V(X,Y)>1 THEN 1230
1210 PRINT " |";:REM SPACE,SHIFT-=
1220 GOTO 1240
1230 PRINT "  ";:REM SPACE,SPACE
1240 NEXT X
1250 PRINT
1255 REM PRINT THE LEFTMOST INTERSECTION
1256 REM THEN A WALL OR OPENING AND ANOTHER INTERSECTION.
1257 PRINT "";:REM CTRL-S
1260 FOR X=1 TO WIDE
1270 IF V(X,Y)=0 THEN 1310
1280 IF V(X,Y)=2 THEN 1310
1290 PRINT " ";:REM SPACE,CTRL-S
1300 GOTO 1320
1310 PRINT "";:REM CTRL-R,CTRL-S
1320 NEXT X
```



```
1330 PRINT
1340 NEXT Y
1344 REM DISPLAY THE COMPLETED MAZE
1345 SETCOLOR 1,12,10
1350 RETURN
1360 REM
2000 REM SUBRT NAME: MAKEBUILDER
2010 REM
2020 REM PERFORMS A RANDOM WALK THROUGH
2030 REM UNMARKED CELLS, KNOCKING DOWN
2040 REM WALLS AS IT GOES. PRINTING
2050 REM INFORMATION IS STORED FOR LATER.
2060 REM
2070 REM PICK A STARTING POINT
2080 X=INT(RND(0)*WIDE)+1
2090 STARTX=2*X
2100 STARTY=1
2110 C=1
2120 W(X,1)=C
2130 Y=1
2140 REM LIST DIRECTIONS AVAILABLE
2150 J=0
2160 IF W(X-1,Y)<>0 THEN 2190
2170 J=J+1
2180 JUMP(J)=XMINUS1
2190 IF W(X+1,Y)<>0 THEN 2220
2200 J=J+1
2210 JUMP(J)=XPLUS1
2220 IF W(X,Y-1)<>0 THEN 2250
2230 J=J+1
2240 JUMP(J)=YMINUS1
2250 IF W(X,Y+1)<>0 THEN 2280
2260 J=J+1
2270 JUMP(J)=YPLUS1
2280 REM IF BOXED IN
2290 IF J=0 THEN 2420
2300 REM SELECT ONE DIRECTION
2310 IF J=1 THEN GOSUB JUMP(1):GOTO 2350
2330 ON INT(RND(0)*J)+1 GOSUB JUMP(1),JUMP(2),JUMP(3)
2340 REM MARK NEW CELL USED
2350 C=C+1
2360 W(X,Y)=C
2370 GOTO 2150
2380 REM IF BOXED IN, START A NEW PATH
2390 REM FROM ANY EXISTING PATH. STOP
2400 REM IF 95% COMPLETE
2420 IF C>0.95*(HIGH*WIDE+1) THEN 2540
2440 X=INT(RND(0)*WIDE)+1
2450 Y=INT(RND(0)*HIGH)+1
2460 IF W(X,Y)<>0 THEN 2150
2470 GOTO 2440
2500 REM OPEN THE BOTTOM OF A MARKED
2510 REM CELL IN THE LAST ROW.
2540 X=INT(RND(0)*WIDE)+1
```

```
2550 IF W(X,HIGH)=0 THEN 2540
2560 V(X,HIGH)=V(X,HIGH)+1
2570 RETURN
3000 REM MAZERUNNER
3005 REM
3010 REM MOVES THE CURSOR ACCORDING
3020 REM TO THE JOYSTICK, CHECKS FOR
3030 REM DONE AND TIMES THE RUN.
3040 REM
3050 REM PUT THE CURSOR AT THE START
3060 X=STARTX:Y=STARTY
3070 COLOR CURSOR:PLOT STARTX,STARTY
3075 REM READ THE CLOCK
3080 A=PEEK(18)
3090 B=PEEK(19)
3100 C=PEEK(20)
3120 FIRSTTIME=(A*256*256+B*256+C)/60
3125 REM READ THE JOYSTICK
3130 D=15-STICK(0)
3140 IF D=0 THEN 3130
3150 X1=-1
3160 IF D<3 THEN X1=0
3170 IF D>7 THEN X1=1
3180 Y1=-1
3190 IF D=INT(D/2)*2 THEN Y1=1
3200 IF D=INT(D/4)*4 THEN Y1=0
3230 REM CHECK FOR WALLS THERE
3240 REM IGNORE JOYSTICK INPUT IF SO
3250 LOCATE X+X1,Y+Y1,CHAR
3260 IF CHAR<>BLANK AND CHAR<>SPOT THEN 3130
3295 REM LEAVE A SPOT, MOVE THE CURSOR
3300 COLOR SPOT:PLOT X,Y
3305 X=X+X1:Y=Y+Y1
3310 COLOR CURSOR:PLOT X,Y
3315 REM REPEAT IF STILL INSIDE
3320 IF Y<BOTTOM THEN 3130
3325 REM READ THE CLOCK AGAIN
3330 A=PEEK(18)
3340 B=PEEK(19)
3350 C=PEEK(20)
3360 LASTTIME=(A*256*256+B*256+C)/60
3370 TIME=INT((LASTTIME-FIRSTTIME)*100)/100
3430 RETURN
4200 REM SUBRT NAME: XMINUS1
4205 REM MOVE LEFT, OPEN NEW CELL'S
4206 REM RIGHT WALL
4210 X=X-1
4220 V(X,Y)=V(X,Y)+2
4230 RETURN
4240 REM
4300 REM SUBRT NAME: YMINUS1
4305 REM MOVE UP, OPEN NEW CELL'S
4306 REM BOTTOM WALL
4310 Y=Y-1
```



```
4320 V(X,Y)=V(X,Y)+1
4330 RETURN
4340 REM
4400 REM SUBRT XPLUS1
4410 REM OPEN THIS CELL'S RIGHT WALL,
4420 REM MOVE RIGHT
4430 V(X,Y)=V(X,Y)+2
4440 X=X+1
4450 RETURN
4460 REM
4500 REM SUBRT NAME: YPLUS1
4510 REM OPEN THIS CELL'S BOTTOM WALL,
4520 REM MOVE DOWN
4530 V(X,Y)=V(X,Y)+1
4540 Y=Y+1
4550 RETURN
```

Monster Combat

Lee Chapel

Translated for the Atari by Shelia Spencer

This monstrous program offers hours of fun. Think carefully before accepting its offer.

Monster Combat is a game in which you go wandering through a forest trying to win as much treasure as you can from various monsters without getting yourself killed in the process. It was written in Basic for a KIM microprocessor and for display on a high speed video board, but can easily be converted to almost any other Basic or video board. It requires at least 16K of RAM to be run, which is the main reason there are no spaces between commands on the program listing.

Lee Chapel, 2349 Wigging, Springfield, Ill. 62704.

Shelia Spencer, Rt. 8, Orchard Hills, 4225 Beulah Cove, Claremore, OK 74017.

Play

When you play the game you will be randomly placed in a forest ten by ten squares in size. Only one of these squares, the one you are in, is displayed, thus allowing you to see only a small part of the forest at a time. The sector you are in is again divided into ten by ten squares. Each of these, too, is divided up to ten by ten; but these hundred smallest squares you see. Each of these little squares is shown by a single claracter. It covers an area of forest ten by ten yards, making the fuller square that is displayed a hundred by a hundred yards and the entire forest a thousand by a thousand yards. T's are trees, '-'s are paths, I's are walls, 's are inns, and M's are enchanted castles. The '0' is you.

Also displayed with the portion of forest you are in is your combat strength, treasure total, and the various magic spells you have. Your combat strength is used to fight the various monsters you meet, each monster having a combat strength of his own; these range from five (for a goblin) to a hundred (for a basilisk). Your combat strength is also used in movement, the amount used depending upon how far you go, how much treasure you're lugging around, and the type of terrain you end up on after you move.

At the inns you are allowed to regain the strength you began with and all the magic you had at the start. Don't worry when you find yourself displayed in the square below the inn when you stop there; that is the way the program is set up. Of course, the innkeeper takes some of your treasure for providing you with his services. However, sometimes he has information which he passes on to you at no additional cost — like where the forest edge is, or where an enchanted castle might be found.

There may be up to fifteen enchanted castles in the forest. These usually contain items of great value to treasure hunters, as you will see. (However, they tend to vanish if you make the wrong move, such as falling into a pit when you land on the castle square.)

Cast of Characters

The following is a description of each monster, giving its combat strength and telling something about the tales and myths surrounding it.

Goblin (5) — A mischievous little sprite only about a yard in height. Rather ugly, uses coarse and uncouth language, is generally evil and malicious; all in all, a rather unpleasant little fellow. Even though they're little they can be very vicious, and more than one warrior has been killed underestimating them.

Minotaur (10) — From Greek mythology, a monster with the head of a bull and the body of a man. Minos, king of Crete, received a bull from Poseidon, god of the sea, which he refused to sacrifice to the god. Poseidon inspired an unnatural love for the bull in Pasiphae, Minos' wife, and the minotaur resulted from the union. Minos enclosed the creature in a labyrinth constructed in the city of Knossos, and fed it seven young men and women (whom Athens had to pay as tribute to Crete) every few years. The original minotaur was eventually slain by the Athenian hero Theseus.

Cyclops (20) — Also from Greek mythology, a member of a race of one-eyed giants. According to Homer, the cyclopses were shepherds living on an island in the western area. The best known of these was Polyphemus who had his eye poked out by the hero, Odysseus. According to Hesiod, the cyclopses were three of the children of Uranus and Gaea. They forged the thunderbolt for Zeus, king of the gods, and became the assistants of Hephaestus, god of the forge.

Zombie (30) — From legends in the West Indies, a corpse which has been reanimated. A rather unpleasant person to meet, he generally smells of rot and decay. He often has rotting pieces of himself falling off his body, yet never seems to fall apart completely. He is difficult to kill, since he is already dead. A person has to chop him into tiny pieces and then get away before the monster can pull himself back together.

Giant (40) — Appears in the mythology of almost all nations, huge beings of terrible aspect. In the Greek myths the giants are said to live in volcanic regions where they were banished after an unsuccessful war against the gods. Some giants are peaceful, but others, like the ones in the forest, would think nothing of having you or anyone else for a snack.

Harpy (50) — From Greek mythology, disgusting women with the wings and lower body of a bird, generally a bird of prey. They stole and befouled the food of blind Phineus as punishment from the gods. Phineus nearly died before Jason and the Argonauts arrived while sailing in search of the Golden Fleece. Two of the Argonauts, Zetes and Calais, drove the harpies away and were then told by one of the gods that the harpies would bother Phineus no more. The harpies continued their disgusting practices elsewhere.

Griffin (60) — From Eastern mythology, a creature usually represented as having the head, beak, and wings of an eagle, and the body and legs of a lion. It builds its nest of gold, making it very tempting to hunters and forcing the griffin to keep vigilant guard. It instinctively knows where buried treasure is hidden and does its best to keep any plunderers at a distance.

Chimera (70) — From Greek mythology, a monster with the foreparts of a lion, the rearparts of a goat with a goat's head in the middle of its back, and with a serpent for a tail. The original chimera was slain by Bellerophon, who was riding on Pegasus, the winged horse. Ironically, Pegasus was a distant relative of the chimera.

Dragon (80) — Found in many of the world's mythologies, a reptilic monster resembling a giant lizard and usually represented as having wings, huge claws, and a fiery breath. In some places the dragon is considered to be a peaceful creature, notably in Japan and China, where it is regarded as a symbol of good fortune. However, the dragons in the forest are of the other sort; they will kill and eat you if you let them, and they take very unkindly to anyone trying to steal their treasure.

Wyvern (90) — A distant relative of the dragon, this is a fabulous two-legged creature, with wings and the head of a dragon on a basilisk's body. Although he cannot kill you with one glance like the basilisk, he is still a very unpleasant creature to meet.

Basilisk (100) — The worst of all eleven monsters, his deadly glare kills anyone who gazes upon his face. From Greek mythology, the basilisk was called the king of serpents, being endowed with a scaly crest upon his head like a crown. This monster was supposedly produced from the egg of a cock hatched under toads or serpents. The weasel, the only animal which can withstand the basilisk's glare, often fought it to the death. Humans must use a mirror if they wish to be assured of victory over a basilisk, for the mirror will reflect the creature's gaze back upon it and kill it. This monster is not to be confused with the basilisk of South America, a harmless lizard with the ability to run across water.

Most of the time you will not be visiting inns and castles. You will be hacking your way through thick underbrush or trotting along forest paths in search of treasure. And you will find it, usually guarded by some sort of monster. Upon encountering one or more of these creatures you are given a choice of fighting them, running away, bribing them, or casting a spell on them.

To fight you must hit a '1'; then, when it asks you to, you enter however much of your combat strength you wish to use against the monster. If you choose to use strength equal to the monster's strength you then have a fifty-fifty chance of winning. The more strength you use the greater the odds are of winning, the less you use the smaller your odds of winning. Also affecting what you use to fight the monster is your treasure total. The more treasure you have the more strength you must use.

The first and third parts of the sample run give examples of fighting a monster or monsters. In the first case there are three cyclopses. Cyclopses have a combat strength of 20 which means that three of them have a total strength of 60. I used 121 of my combat strength to fight them, over twice the cyclopses' strength, which gave me over a 95% chance of winning. And, as can be seen in the example, I did beat him.

In the third part of the sample run I am fighting 19 goblins. Since goblins have a combat strength of 5, 19 have a combined strength of 95. I used only 60 combat points that time, giving me around a 30% chance of winning. And, as can be seen in the example, I did get myself killed.

If you do not wish to fight the monster you can always run. However, the higher the strength of the monster the less likely you will get away and the more likely that you will be forced to fight. Whether or not you do get away is based upon a random number and the strength of the monster. If you do get away you are randomly placed in an adjacent square and get to find out what is there. Once in a while, when you attempt to run, the monster catches you and kills you.

If you don't care to run or fight you can try to bribe the monster. Few people like to do this since it means handing over some of your hard-earned treasure. Whether your bribe is accepted or not depends upon how much treasure the monster is guarding, his strength, and a random number. The greater the value of the treasure the monster has, the more you'll have to pay him if you don't care to fight. Usually if the monster doesn't care for your bribe you have to fight him. Sometimes, though, he just kills you anyway.

Finally, if you don't care for any of the previous choices, you may cast a spell. There are three types of spells: sleep, charms, and invisibility. Sleep spells tend to be the least effective and invisibility the most effective, with charms somewhere in the middle. Spells, no matter what kind they are, don't always work too well, sometimes not working at all, thus causing you to have to fight the monster.

In addition to the various monsters, there are other things you will occasionally run into; some are good and some bad, as you will see when you run the program. Everything is determined randomly and thus you can go back to a spot you were previously at and find something different there.

You have thirty days to hunt for treasure in the forest. Each little square you move through takes a tenth of a day to cross, meaning it takes an entire day to cross the entire displayed square. To move you enter the direction you wish to go (N meaning North, which is upwards, S meaning South, E meaning East, which is to the right, and W meaning West). Then you enter the distance, each little square being one. For example, in the first part of the sample run I enter S (south) for the direction and then 3 for the distance. This places me on top of the arrow, which is an inn, and thus I am shown in the square below the inn when the next map of the area is drawn. In moving from the inn I again go south, this time a distance of 7, which causes me to end up in the next large square.

When you leave the forest, intentionally or accidentally, you can obtain a listing of the number of monsters you've killed, bribed and run from, plus the amount of treasure you have won so far. If you decide not to return to the forest or your thirty days are up, you are offered several choices: you may go to a new forest with the same strength and magic (the treasure total going back to zero); you may go to a new forest with new strength and magic; or you can stop playing the game. If you should wish to use the strength and magic left over from the game you just played, you can obtain a listing of these at the very end of the game and then write them down or store them however you wish. Then, the next time you play the game, you just answer the initial question asking if you wish to use an old combat strength and magic with a 'Y' and then enter the various things you are asked for.

This game was very popular at my dorm at the University of Wisconsin in Madison. The record treasure total so far, as of this writing, is 7562, set by me. Most of the time the scores run between a thousand and two thousand, with many lower and a few higher. If you get above two thousand you're doing well.

Menu from Monster Disk

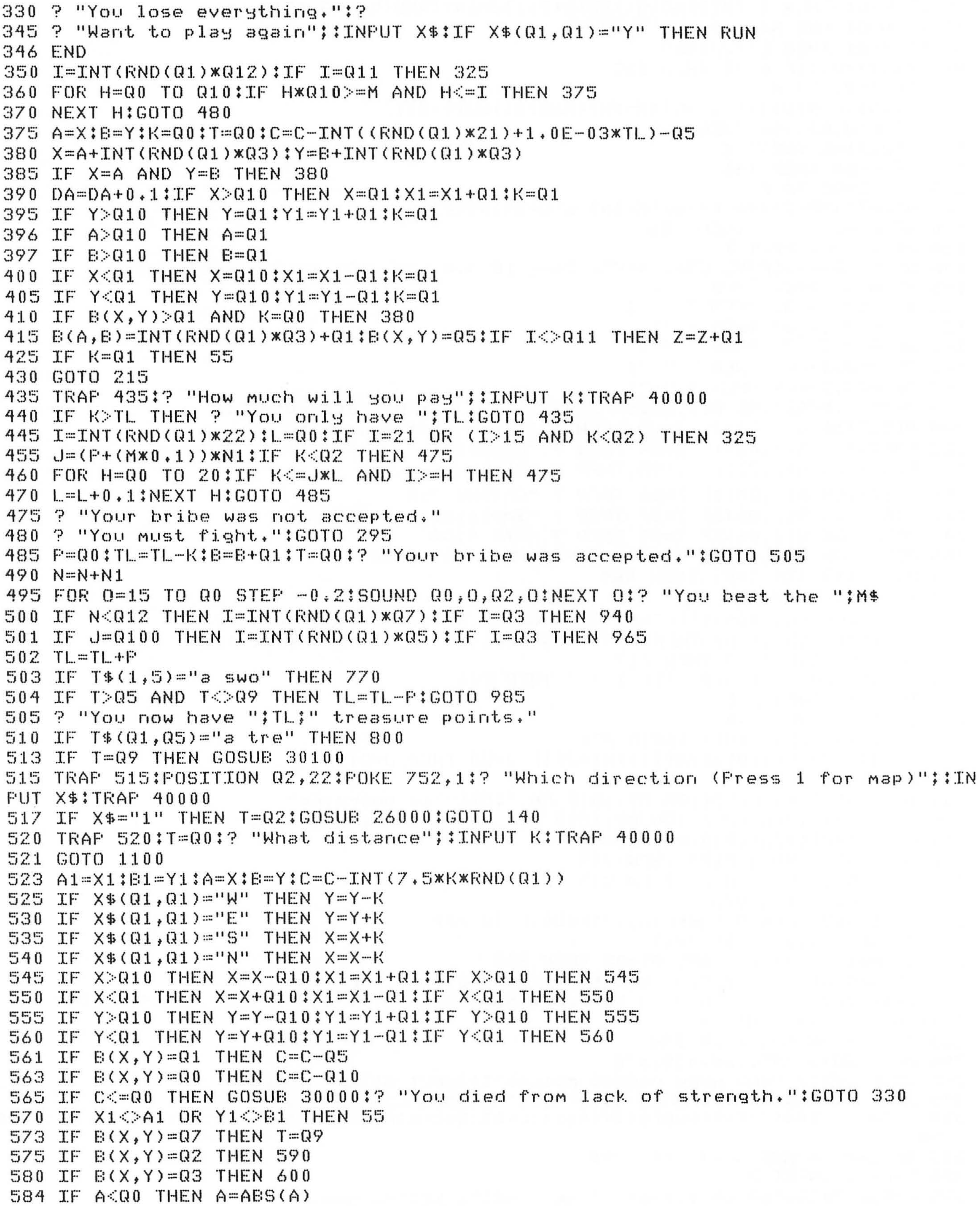
```
0 GRAPHICS 0
1 SETCOLOR 2,0,10:SETCOLOR 1,0,2:SETCOLOR 4,0,10:POKE 82,2:POKE 83,
  39:POKE 752,1
2 ? "}          ***DIRECTORY***":?
6 GOTO 9
8 FOR I=1 TO 1000:NEXT I:RETURN
9 DIM X$(40),L$(20),A$(40),P$(500),N$(2),B0$(40)
10 REM "D1:MENU"
11 OPEN #1,6,0,"D:*.*"
12 B0$="                                  ":GOTO 30
20 N=N+1:X$=A$(3,10):X$(9,9)=".":X$(10,12)=A$(11,13)
22 L$=" ":L$(LEN(L$)+1)=X$:P$(LEN(P$)+1)=X$
24 L$(14,14)="(":FOR I=2 TO LEN(L$):IF L$(I,I)=" " THEN L$(I,I)="."
25 NEXT I:N$=STR$(N):IF N<10 THEN N$(2,2)=N$(1,1):N$(1,1)="0"
27 RETURN
30 INPUT #1,A$
40 IF A$(2,2)<>" " THEN GOTO 90
45 GOSUB 20
50 ? L$;N$;") ";
60 INPUT #1,A$
70 IF A$(2,2)<>" " THEN GOTO 90
```

```
75 GOSUB 20
80 ? L$(2,14);N$;") "
85 GO TO 30
90 ? :? "            ";A$
120 POKE 752,0
130 POSITION 1,22:? "  SELECTION";:TRAP 130:INPUT X:? "";:TRAP 40000
131 IF X<>INT(X) THEN 130
135 IF X=1 THEN X$=P$(1,12):GOTO 145
140 X$=P$((X-1)*12+1,(X-1)*12+12):TRAP 40000
141 IF X$(11,11)=" " THEN X$=X$(1,8)
145 IF X$(1,3)="DOS" THEN DOS
150 A$="D1:":A$(LEN(A$)+1)=X$
155 POKE 752,3:POSITION 1,22:PRINT "             LOADING ";X$;
160 TRAP 200:RUN A$:TRAP 40000
200 POSITION 1,22:PRINT "  CANNOT RUN ";X$;:GOSUB 8:TRAP 40000:GOTO 130
```

Monster

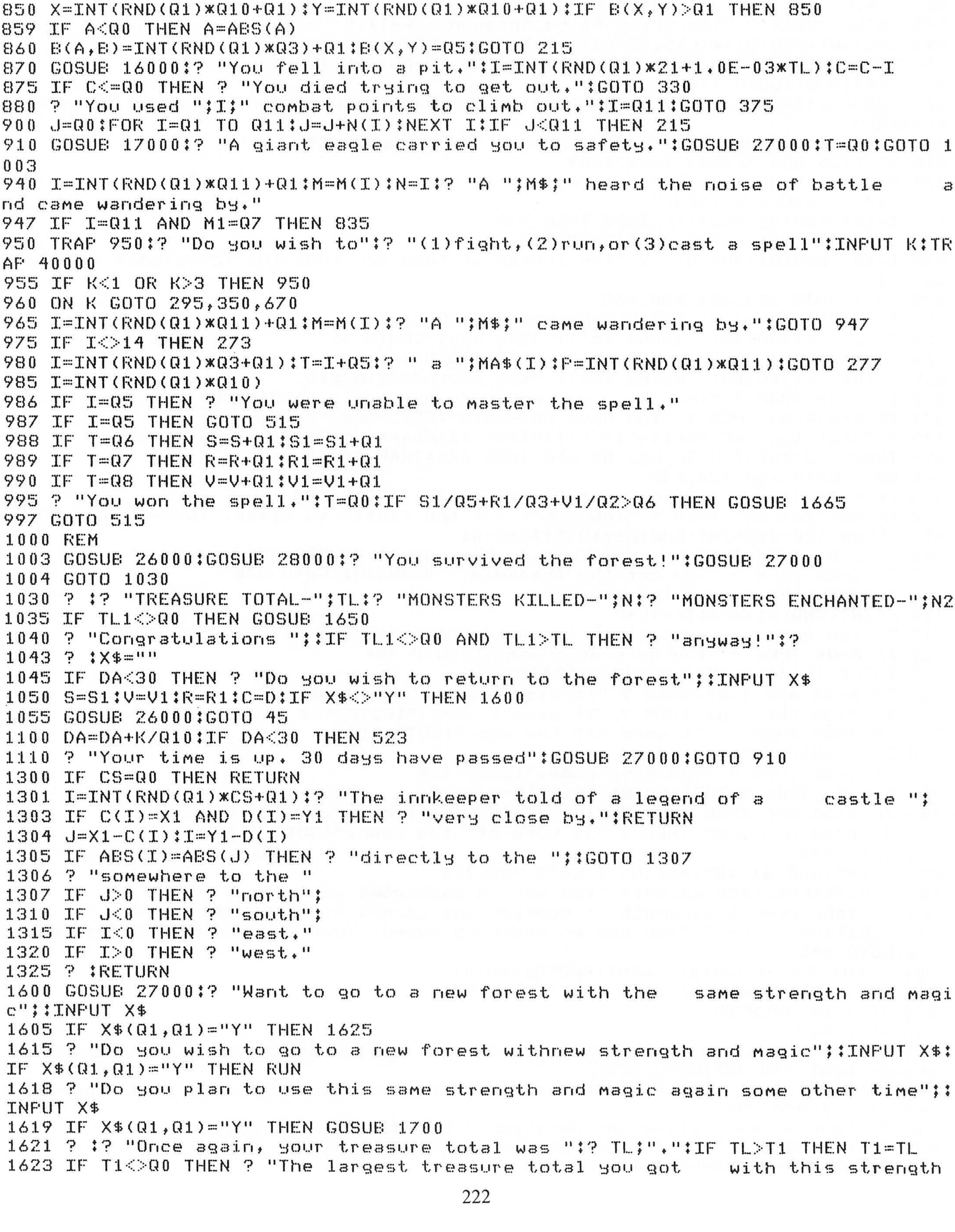
```
0 GRAPHICS 2:POKE 752,1:POSITION 6,4:? #6;"MONSTER":POSITION 6,6:? #6;"COMBAT":?
 "                By Lee Chapel"
1 ? "Translated for ATARI by Sheila Spencer":GOSUB 27000:GRAPHICS 0:GOSUB 26000
3 POKE 559,0:Q0=0:Q1=1:Q2=2:Q3=3:Q4=4:Q5=5:Q6=6:Q7=7:Q8=8:Q9=9:Q10=10:Q11=11:Q12
=12:Q100=100:Q10000=10000:N=Q0
4 POKE 752,Q1:DIM A(Q10,Q10),B(Q10,Q10),M(Q11),M$(Q8),N(Q11),T$(50),Z(Q11),MA$(2
0),X$(Q3),C(15),D(15),P(Q11)
5 FOR E=Q1 TO Q10:FOR F=Q1 TO Q10:A(E,F)=0:B(E,F)=0:NEXT F:NEXT E
8 GOSUB 26000
9 RESTORE :FOR I=Q1 TO Q11:READ Q:M(I)=Q:NEXT I:FOR I=Q1 TO Q11:READ Z:P(I)=Z:NE
XT I:V=INT(RND(Q1)*Q3)
15 C=INT(RND(Q1)*1501+500):S=INT(RND(Q1)*Q6):R=INT(RND(Q1)*Q4):MA$(Q1)="  Sleep
Spell":MA$(Q2)="  Charm"
17 MA$(Q3)="Invisibility Spell":POKE 559,34:? "Want the strength and magic from
    another game";:INPUT X$
18 IF X$(Q1,Q1)="Y" THEN 1630
20 D=C:V1=V:S1=S:R1=R:? "Just a moment...":FOR Q=1 TO 750:NEXT Q:POKE 559,Q0
25 FOR I=Q1 TO Q10:FOR J=Q1 TO Q10:T=INT(RND(Q1)*Q10):IF T<>Q1 OR CS=15 THEN T=Q
0
26 H=INT(RND(Q1)*Q2):W=INT(RND(Q1)*Q10)
30 P=INT(RND(Q1)*51):A(I,J)=Q10000*T+Q100*P+Q10*W+H
37 IF T=Q1 THEN CS=CS+Q1:C(CS)=I:D(CS)=J
40 NEXT J:NEXT I:T=Q0:SETCOLOR Q2,13,Q6:SETCOLOR Q4,13,Q6:SETCOLOR Q1,13,13
45 X1=INT(RND(Q1)*Q8)+Q2:Y1=INT(RND(Q1)*Q8)+Q2:X=INT(RND(Q1)*Q10)+Q1:Y=INT(RND(Q
1)*Q10)+Q1
55 IF X1<Q1 OR X1>Q10 OR Y1<Q1 OR Y1>Q10 THEN 1000
56 FOR I=Q1 TO Q10:FOR J=Q1 TO Q10:B(I,J)=Q0:NEXT J:NEXT I:CA=INT(A(X1,Y1)/Q1000
0)
57 P=INT((A(X1,Y1)-(Q10000*CA))/Q100)
60 W=INT((A(X1,Y1)-(Q10000*CA)-(Q100*P))/Q10):H=A(X1,Y1)-Q10000*CA-Q100*P-Q10*W:
I=Q0:J=Q0
67 IF CA=Q1 THEN I=INT(RND(Q1)*Q10+Q1):J=INT(RND(Q1)*Q10+Q1):B(I,J)=Q7
70 IF CA=Q1 AND I=X AND Y=J THEN B(I,J)=Q0:GOTO 67
```

```
75 IF H=Q1 THEN I=INT(RND(Q1)*Q10+Q1):J=INT(RND(Q1)*Q9+Q1)
85 IF H=Q1 AND B(I,J)<>Q0 THEN 75
87 IF H=Q1 THEN B(I,J)=Q3
90 B(X,Y)=Q5:IF W=Q0 THEN 115
95 FOR I=Q1 TO W
100 J=INT(RND(Q1)*Q10)+Q1:K=INT(RND(Q1)*Q10+Q1)
105 IF B(J,K)<>Q0 THEN 100
110 B(J,K)=Q2:NEXT I
115 IF P=Q0 THEN 140
120 FOR I=Q1 TO P
125 J=INT(RND(Q1)*Q10)+Q1:K=INT(RND(Q1)*Q10)+Q1
130 IF B(J,K)<>Q0 THEN 125
135 B(J,K)=Q1:NEXT I
140 GOSUB 26000:POKE 559,34:FOR I=Q1 TO Q10:FOR J=Q1 TO Q10:POSITION I,J
145 IF B(J,I)=Q0 THEN ? "C";
150 IF B(J,I)=Q1 THEN ? " ";
155 IF B(J,I)=Q2 THEN ? "";
160 IF B(J,I)=Q3 THEN ? "";
165 IF B(J,I)=Q5 THEN ? "O";
167 IF B(J,I)=Q7 THEN ? "M";
170 NEXT J:POSITION Q12,Q0:IF I=Q2 THEN ? "Combat Strength=";C
180 POSITION Q12,Q1:IF I=Q3 THEN ? "Treasure total=";TL
185 POSITION Q12,Q2:IF I=Q4 THEN ? "Magic:"
190 POSITION Q12,Q3:IF I=Q5 THEN ? "Sleep-";S
195 POSITION Q12,Q4:IF I=Q6 THEN ? "Charms-";R
200 POSITION Q12,Q5:IF I=Q7 THEN ? "Invisibility-";V
203 POSITION Q12,Q6:IF I=Q9 THEN ? "DAY ";DA
205 POSITION Q12,Q7:IF I=Q1 OR I=Q8 OR I=Q10 THEN ?
210 NEXT I:? :IF T=Q1 THEN 604
213 IF T=Q2 THEN 515
215 I=INT(RND(Q1)*Q5):IF I=Q2 THEN GOSUB 32000:POSITION Q0,Q12
220 IF I=Q1 AND T<>Q9 THEN POSITION Q0,14:? "Nothing there.":GOTO 515
223 IF I=Q1 AND T=Q9 THEN 513
225 I=INT(RND(Q1)*16+Q1):IF I=Q12 THEN 840
235 IF I=13 THEN 870
237 IF I=14 THEN 900
240 IF I>14 THEN J=Q100:GOTO 270
245 J=INT(RND(Q1)*Q100/M(I)):N1=J:IF J=Q0 THEN J=Q1:N1=J
254 GOSUB 10000
255 IF J=Q1 THEN POSITION Q2,15:? "A ";M$;" is guarding"
260 IF J<>Q1 THEN POSITION Q2,15:? J;" ";M$;"s are guarding"
265 M=M(I)*J:I=INT(RND(Q1)*14+Q1)
270 IF I>Q11 AND J=Q100 THEN 215
271 IF I<Q12 AND J=Q100 THEN 215
272 IF I>Q11 THEN 975
273 IF I>Q12 THEN ? "nothing.":P=Q0:GOTO 277
275 GOSUB 10025:? T$:P=P(I)
277 IF M$(1,3)="Bas" AND M1=Q7 THEN 835
279 IF J=Q100 THEN ? "You get the treasure free!":GOTO 500
280 TRAP 280:? "Do you wish to (1)fight, (2)run,":? "(3)bribe, or (4)cast a spel
1";:INPUT K:TRAP 40000
285 IF K<1 OR K>4 THEN 280
290 ON K GOTO 295,350,435,670
295 TRAP 295:? "How many combat points";:INPUT K:TRAP 40000
300 IF K>C THEN ? "You only have ";C;" combat points.":GOTO 295
304 GOSUB 14000:I=INT(RND(Q1)*1001):L=Q2:C=C-K:K=K-0.01*TL:FOR H=1000 TO Q0 STEP
 -50
315 IF L*M<=K AND H>=I THEN 490
320 L=L-0.1:NEXT H
325 GOSUB 30000:GOSUB 13000:? "The ";M$;"s killed you."
```



```
330 ? "You lose everything.":?
345 ? "Want to play again";:INPUT X$:IF X$(Q1,Q1)="Y" THEN RUN
346 END
350 I=INT(RND(Q1)*Q12):IF I=Q11 THEN 325
360 FOR H=Q0 TO Q10:IF H*Q10>=M AND H<=I THEN 375
370 NEXT H:GOTO 480
375 A=X:B=Y:K=Q0:T=Q0:C=C-INT((RND(Q1)*21)+1.0E-03*TL)-Q5
380 X=A+INT(RND(Q1)*Q3):Y=B+INT(RND(Q1)*Q3)
385 IF X=A AND Y=B THEN 380
390 DA=DA+0.1:IF X>Q10 THEN X=Q1:X1=X1+Q1:K=Q1
395 IF Y>Q10 THEN Y=Q1:Y1=Y1+Q1:K=Q1
396 IF A>Q10 THEN A=Q1
397 IF B>Q10 THEN B=Q1
400 IF X<Q1 THEN X=Q10:X1=X1-Q1:K=Q1
405 IF Y<Q1 THEN Y=Q10:Y1=Y1-Q1:K=Q1
410 IF B(X,Y)>Q1 AND K=Q0 THEN 380
415 B(A,B)=INT(RND(Q1)*Q3)+Q1:B(X,Y)=Q5:IF I<>Q11 THEN Z=Z+Q1
425 IF K=Q1 THEN 55
430 GOTO 215
435 TRAP 435:? "How much will you pay";:INPUT K:TRAP 40000
440 IF K>TL THEN ? "You only have ";TL:GOTO 435
445 I=INT(RND(Q1)*22):L=Q0:IF I=21 OR (I>15 AND K<Q2) THEN 325
455 J=(P+(M*0.1))*N1:IF K<Q2 THEN 475
460 FOR H=Q0 TO 20:IF K<=J*L AND I>=H THEN 475
470 L=L+0.1:NEXT H:GOTO 485
475 ? "Your bribe was not accepted."
480 ? "You must fight.":GOTO 295
485 P=Q0:TL=TL-K:B=B+Q1:T=Q0:? "Your bribe was accepted.":GOTO 505
490 N=N+N1
495 FOR O=15 TO Q0 STEP -0.2:SOUND Q0,O,Q2,O:NEXT O:? "You beat the ";M$
500 IF N<Q12 THEN I=INT(RND(Q1)*Q7):IF I=Q3 THEN 940
501 IF J=Q100 THEN I=INT(RND(Q1)*Q5):IF I=Q3 THEN 965
502 TL=TL+P
503 IF T$(1,5)="a swo" THEN 770
504 IF T>Q5 AND T<>Q9 THEN TL=TL-P:GOTO 985
505 ? "You now have ";TL;" treasure points."
510 IF T$(Q1,Q5)="a tre" THEN 800
513 IF T=Q9 THEN GOSUB 30100
515 TRAP 515:POSITION Q2,22:POKE 752,1:? "Which direction (Press 1 for map)";:IN
PUT X$:TRAP 40000
517 IF X$="1" THEN T=Q2:GOSUB 26000:GOTO 140
520 TRAP 520:T=Q0:? "What distance";:INPUT K:TRAP 40000
521 GOTO 1100
523 A1=X1:B1=Y1:A=X:B=Y:C=C-INT(7.5*K*RND(Q1))
525 IF X$(Q1,Q1)="W" THEN Y=Y-K
530 IF X$(Q1,Q1)="E" THEN Y=Y+K
535 IF X$(Q1,Q1)="S" THEN X=X+K
540 IF X$(Q1,Q1)="N" THEN X=X-K
545 IF X>Q10 THEN X=X-Q10:X1=X1+Q1:IF X>Q10 THEN 545
550 IF X<Q1 THEN X=X+Q10:X1=X1-Q1:IF X<Q1 THEN 550
555 IF Y>Q10 THEN Y=Y-Q10:Y1=Y1+Q1:IF Y>Q10 THEN 555
560 IF Y<Q1 THEN Y=Y+Q10:Y1=Y1-Q1:IF Y<Q1 THEN 560
561 IF B(X,Y)=Q1 THEN C=C-Q5
563 IF B(X,Y)=Q0 THEN C=C-Q10
565 IF C<=Q0 THEN GOSUB 30000:? "You died from lack of strength.":GOTO 330
570 IF X1<>A1 OR Y1<>B1 THEN 55
573 IF B(X,Y)=Q7 THEN T=Q9
575 IF B(X,Y)=Q2 THEN 590
580 IF B(X,Y)=Q3 THEN 600
584 IF A<Q0 THEN A=ABS(A)
```

```
585 B(A,B)=INT(RND(Q1)*Q3):B(X,Y)=Q5:GOTO 140
590 GOSUB 29000:? "You tried to go through a wall."
595 C=C-INT(RND(Q1)*TL*5.0E-03)-25:X=A:Y=B:GOTO 515
600 Y=Y+Q1:C=D:B(A,B)=INT(RND(Q1)*Q3):B(X,Y)=Q5:T=Q1:V=V1
603 R=R1:S=S1:GOTO 140
604 GOSUB 31000:POSITION Q2,Q12:? "You stopped at an inn and regained    your st
rength."
610 I=INT(RND(Q1)*TL*0.25):IF I<Q5 AND TL>Q5 THEN I=Q5
615 IF I<Q5 AND TL<=Q5 THEN I=Q0
620 ? "You paid ";I;" treasure points to stay    there.":TL=TL-I:? "You now have
 ";TL;" treasure points."
630 I=INT(RND(Q1)*Q3):IF I=Q2 THEN 515
633 IF I=Q1 THEN GOSUB 1300:GOTO 515
635 I=INT(RND(Q1)*Q4+Q1):? "The innkeeper told you that the forestedge is less t
han ";
645 ON I GOTO 650,655,660,665
650 ? Y1*Q100;" yards to":? "the west":GOTO 515
655 ? (Q11-X1)*Q100;" yards to":? "the east":GOTO 515
660 ? X1*Q100;" yards to":? "the north":GOTO 515
665 ? (Q11-X1)*Q100;" yards to":? "the south":GOTO 515
670 IF T>Q5 THEN ? "You can't use magic to get magic.":GOTO 280
671 IF S+V+R=Q0 THEN ? "You have no magic":GOTO 280
673 ? "What type of spell-":? "(1)Sleep,(2)Charm, or (3)Invisibility";
675 TRAP 673:INPUT K:IF K<1 OR K>3 THEN 670:TRAP 40000
680 ON K GOTO 685,720,745
685 IF S=Q0 THEN ? "You have no Sleep Spells.":GOTO 480
690 IF M$="Zombie" THEN ? "You can't put the ";M$;" to sleep.":S=S-Q1:GOTO 480
695 GOSUB 12000:I=INT(RND(Q1)*Q10):S=S-Q1
700 IF I<Q3 THEN ? "Your spell was unsuccessfull.":GOTO 480
705 IF I<Q8 THEN ? "You got the treasure.":N2=N2+N1:GOTO 500
710 ? "The ";M$;" woke too soon."
713 P=INT(RND(Q1)*P):TL=TL+P
715 ? "You got away with ";P;" treasure points":N2=N2+N1:GOTO 515
720 IF R=Q0 THEN ? "You have no charms.":GOTO 480
725 GOSUB 12000:I=INT(RND(Q1)*Q10):R=R-Q1
730 IF M<60 AND I>Q6 THEN ? "It didn't work.":GOTO 480
733 IF M>50 AND I<Q2 THEN ? "It didn't work.":GOTO 480
735 IF I=Q3 THEN ? "It wore off too soon":GOTO 713
740 I=Q3:GOTO 705
745 IF V=Q0 THEN ? "You have none.":GOTO 480
750 GOSUB 12000:I=INT(RND(Q1)*Q10):V=V-Q1
755 IF M>50 AND I>Q8 THEN ? "The ";M$;" smelled you.":GOTO 713
760 IF M<60 AND I=Q0 THEN ? "It wore off too soon.":GOTO 713
765 GOTO 740
770 I=INT(RND(Q1)*Q2)+Q1:ON I GOTO 780,790
780 C=Q2*C:POSITION Q2,Q4:? "You won an enchanted sword."
781 ? "Your combat strength is doubled and is now ";C;".":GOTO 505
790 POSITION Q2,Q4:? "You won an ordinary sword. Your combatstrength remains at
";C:GOTO 505
800 J=INT(RND(Q1)*Q10):I=INT(RND(Q1)*Q10)
805 IF J=Q7 AND M1<>Q7 THEN M1=Q7:GOTO 820
810 IF I=Q1 THEN 830
815 GOTO 513
820 ? "There was a mirror in the chest. It   will protect you against any Basili
sksyou meet.":M1=Q7:GOTO 515
830 GOSUB 11000:? "The treasure chest was a trap. You    were killed when you op
ened it.":GOTO 330
835 ? "Your mirror killed the Basilisk!":N=N+J:M=Q0:GOTO 500
840 GOSUB 15000:? "A giant bat grabbed you and carried   you to a new spot."
845 A=X:B=Y:T=Q0:DA=DA+0.1
```



```
850 X=INT(RND(Q1)*Q10+Q1):Y=INT(RND(Q1)*Q10+Q1):IF B(X,Y)>Q1 THEN 850
859 IF A<Q0 THEN A=ABS(A)
860 B(A,B)=INT(RND(Q1)*Q3)+Q1:B(X,Y)=Q5:GOTO 215
870 GOSUB 16000:? "You fell into a pit.":I=INT(RND(Q1)*21+1.0E-03*TL):C=C-I
875 IF C<=Q0 THEN ? "You died trying to get out.":GOTO 330
880 ? "You used ";I;" combat points to climb out.":I=Q11:GOTO 375
900 J=Q0:FOR I=Q1 TO Q11:J=J+N(I):NEXT I:IF J<Q11 THEN 215
910 GOSUB 17000:? "A giant eagle carried you to safety.":GOSUB 27000:T=Q0:GOTO 1
003
940 I=INT(RND(Q1)*Q11)+Q1:M=M(I):N=I:? "A ";M$;" heard the noise of battle      a
nd came wandering by."
947 IF I=Q11 AND M1=Q7 THEN 835
950 TRAP 950:? "Do you wish to":? "(1)fight,(2)run,or(3)cast a spell":INPUT K:TR
AP 40000
955 IF K<1 OR K>3 THEN 950
960 ON K GOTO 295,350,670
965 I=INT(RND(Q1)*Q11)+Q1:M=M(I):? "A ";M$;" came wandering by.":GOTO 947
975 IF I<>14 THEN 273
980 I=INT(RND(Q1)*Q3+Q1):T=I+Q5:? " a ";MA$(I):P=INT(RND(Q1)*Q11):GOTO 277
985 I=INT(RND(Q1)*Q10)
986 IF I=Q5 THEN ? "You were unable to master the spell."
987 IF I=Q5 THEN GOTO 515
988 IF T=Q6 THEN S=S+Q1:S1=S1+Q1
989 IF T=Q7 THEN R=R+Q1:R1=R1+Q1
990 IF T=Q8 THEN V=V+Q1:V1=V1+Q1
995 ? "You won the spell.":T=Q0:IF S1/Q5+R1/Q3+V1/Q2>Q6 THEN GOSUB 1665
997 GOTO 515
1000 REM
1003 GOSUB 26000:GOSUB 28000:? "You survived the forest!":GOSUB 27000
1004 GOTO 1030
1030 ? :? "TREASURE TOTAL-";TL:? "MONSTERS KILLED-";N:? "MONSTERS ENCHANTED-";N2
1035 IF TL1<>Q0 THEN GOSUB 1650
1040 ? "Congratulations ";:IF TL1<>Q0 AND TL1>TL THEN ? "anyway!":?
1043 ? :X$=""
1045 IF DA<30 THEN ? "Do you wish to return to the forest";:INPUT X$
1050 S=S1:V=V1:R=R1:C=D:IF X$<>"Y" THEN 1600
1055 GOSUB 26000:GOTO 45
1100 DA=DA+K/Q10:IF DA<30 THEN 523
1110 ? "Your time is up. 30 days have passed":GOSUB 27000:GOTO 910
1300 IF CS=Q0 THEN RETURN
1301 I=INT(RND(Q1)*CS+Q1):? "The innkeeper told of a legend of a     castle ";
1303 IF C(I)=X1 AND D(I)=Y1 THEN ? "very close by.":RETURN
1304 J=X1-C(I):I=Y1-D(I)
1305 IF ABS(I)=ABS(J) THEN ? "directly to the ";:GOTO 1307
1306 ? "somewhere to the "
1307 IF J>0 THEN ? "north";
1310 IF J<0 THEN ? "south";
1315 IF I<0 THEN ? "east."
1320 IF I>0 THEN ? "west."
1325 ? :RETURN
1600 GOSUB 27000:? "Want to go to a new forest with the   same strength and magi
c";:INPUT X$
1605 IF X$(Q1,Q1)="Y" THEN 1625
1615 ? "Do you wish to go to a new forest withnew strength and magic";:INPUT X$:
IF X$(Q1,Q1)="Y" THEN RUN
1618 ? "Do you plan to use this same strength and magic again some other time";:
INPUT X$
1619 IF X$(Q1,Q1)="Y" THEN GOSUB 1700
1621 ? :? "Once again, your treasure total was ":? TL;".":IF TL>T1 THEN T1=TL
1623 IF T1<>Q0 THEN ? "The largest treasure total you got    with this strength
```

```
and magic was":? T1;"."
1624 ? "You killed ";N;" monsters.":? "You successfully worked magic on":? N2;"
monsters.":END
1625 B=Q0:Z=Q0:DA=Q0:FOR I=Q1 TO Q11:N(I)=Q0:NEXT I:IF Q1<TL THEN T1=T'
1627 TL=Q0:GOTO 20
1630 ? "COMBAT STRENGTH";:INPUT C:IF C<500 OR C>2000 THEN 1630
1640 ? "SLEEP SPELLS";:INPUT S:? "CHARMS";:INPUT R:? "INVISIBILITY";:INPUTV:? "
PREVIOUS LARGEST TOTAL";
1645 INPUT T1:GOTO 20
1650 IF T1<TL THEN ? "You won more treasure this time than  before."
1653 IF T1>TL THEN ? "You didn't get as much treasure this  time."
1660 RETURN
1665 ? "Your magic total is rather large.":? "Do you wish to convert it to comba
t":? "points";:INPUT X$
1670 IF X$(Q1,Q1)="N" THEN RETURN
1675 S1=S1-Q5:R1=R1-Q3:V1=V1-Q2:IF S1<=Q0 THEN S1=Q1
1680 IF R1<=Q0 THEN R1=Q1
1685 IF V1<=Q0 THEN V1=Q1
1690 S=S1:R=R1:V=V1:C=C+Q100:D=D+Q100:? "Your combat strength is permanently   i
ncreased by 100.":RETURN
1700 ? "COMBAT STRENGTH-";D:? "SLEEP SPELLS-";S1:? "CHARMS-";R1:? "INVISIBILITY-
";V1:? :RETURN
2000 DATA 5,10,10,25,20,50,30,100,40,50,50,200,60,50,70,30,80,75,90,100,100,50
9000 END
10000 I=INT(RND(Q1)*Q11)+Q1
10002 IF I=Q1 THEN M$="Goblin"
10004 IF I=Q2 THEN M$="Minotaur"
10006 IF I=Q3 THEN M$="Cyclops"
10008 IF I=Q4 THEN M$="Zombie"
10010 IF I=Q5 THEN M$="Giant"
10012 IF I=Q6 THEN M$="Harpy"
10014 IF I=Q7 THEN M$="Griffin"
10016 IF I=Q8 THEN M$="Chimera"
10018 IF I=Q9 THEN M$="Dragon"
10020 IF I=Q10 THEN M$="Wyvern"
10022 IF I=Q11 THEN M$="Basilisk"
10024 RETURN
10025 I=INT(RND(Q0)*Q11)+Q1
10030 IF I=Q1 THEN T$="10 silver spoons (10 pts)"
10032 IF I=Q2 THEN T$="a sword which might be enchanted       (25 pts)"
10034 IF I=Q3 THEN T$="50 silver coins (50 pts)"
10036 IF I=Q4 THEN T$="100 gold pieces (100 pts)"
10038 IF I=Q5 THEN T$="an emerald bracelet (50 pts)"
10040 IF I=Q6 THEN T$="a treasure chest (200 pts)"
10042 IF I=Q7 THEN T$="a pearl necklace (50 pts)"
10044 IF I=Q8 THEN T$="a jeweled sword (30 pts)"
10046 IF I=Q9 THEN T$="a jar of rubies (75 pts)"
10048 IF I=Q10 THEN T$="a box of jewels (100 pts)"
10050 IF I=Q11 THEN T$="a gold goblet (50 pts)"
10055 RETURN
11000 V=15:FOR O=Q10 TO Q100:SOUND Q1,O,Q0,V:SETCOLOR Q2,O,V:V=0.99*V:NEXT O:FOR
 Z=Q0 TO Q10:NEXT Z
11005 SOUND Q1,Q0,Q0,Q0:SETCOLOR Q2,Q9,Q1:GOSUB 26000:RETURN
12000 GOSUB 26000:POSITION 12,12:SETCOLOR Q1,Q5,Q1:? "MAGIC BEING USED...":SOUND
 Q1,Q0,Q0,Q3:SOUND Q0,49,Q10,Q5
12001 SETCOLOR Q2,Q1,14:SETCOLOR Q4,Q1,14:FOR O=1 TO 75:NEXT O:SOUND Q0,Q0,Q0,Q0
:SOUND Q0,25,Q10,Q5
12005 SETCOLOR Q2,Q4,14:SETCOLOR Q4,Q4,14:FOR O=Q1 TO 50:NEXT O
12006 SOUND Q0,Q0,Q0,Q0
12010 SOUND Q0,26,Q10,Q5:SETCOLOR Q2,Q5,Q10:SETCOLOR Q4,Q5,Q10:FOR O=Q1 TO 50:NE
```

```
XT O:SOUND Q0,Q0,Q0,Q0
12011 SOUND Q1,Q0,Q0,Q0
12015 GOSUB 26000:RETURN
13000 SETCOLOR Q2,Q0,Q0:SETCOLOR Q4,Q0,Q0:RESTORE 13015:FOR Z=Q1 TO 15:READ NTE:
READ DUR
13005 SOUND Q0,NTE,Q10,Q5:FOR Q=Q1 TO DUR:NEXT Q:SOUND Q0,Q0,Q0,Q0
13010 IF NTE=Q0 AND DUR=Q0 THEN SOUND Q0,Q0,Q0,Q0
13011 NEXT Z
13015 DATA 136,8,136,3,102,40,136,9,102,3,85,36,136,8,102,3,85,20,136,8,102,3,85
,20,136,8,102,3,85,56,0,0
13020 GOSUB 26000:SETCOLOR Q2,Q9,Q1:RETURN
14000 FOR O=155 TO Q0 STEP -2:SOUND Q0,Q10,Q0,O:NEXT O:SOUND Q0,Q0,Q0,Q0
14005 GOSUB 26000:RETURN
15000 FOR O=255 TO Q0 STEP -3:SOUND Q0,O,O,O:NEXT O:GOSUB 26000:RETURN
16000 FOR O=Q0 TO 255 STEP Q7:SOUND Q0,O,Q2,O:NEXT O:GOSUB 26000:RETURN
17000 FOR O=15 TO Q0 STEP -0.25:SOUND Q0,50,Q10,O:NEXT O:GOSUB 26000:RETURN
19000 FOR O=Q0 TO 15 STEP 0.5:SOUND Q0,O,O,O:NEXT O
19010 FOR O=15 TO 0 STEP -0.5:SOUND 0,O,O,O:NEXT O:GOSUB 26000:RETURN
21000 FOR O=Q0 TO Q8:FOR T=Q1 TO Q8:SOUND Q0,Q10,O,O:SOUND Q1,T,T,T:NEXT T:NEXT
O:SOUND Q0,Q0,Q0,Q0
21005 SOUND Q1,Q0,Q0,Q0:GOSUB 26000:RETURN
22000 FOR O=15 TO Q0 STEP -2:SOUND Q0,Q10,Q8,O:NEXT O:SOUND Q0,Q0,Q0,Q0:GOSUB 26
000:RETURN
23000 FOR O=Q0 TO 15:SOUND Q0,Q10,O,O:NEXT O:GOSUB 26000:RETURN
24000 FOR O=Q0 TO 255 STEP Q8:SOUND Q0,O,Q2,O:NEXT O:GOSUB 26000:RETURN
25000 SETCOLOR 2,5,8:SETCOLOR 4,5,8:FOR O=89 TO 24 STEP -10:SOUND 0,O,10,8:SOUND
 1,O+10,10,8
25001 SOUND 2,O+20,10,8:SOUND 3,O+30,10,8:NEXT O:FOR VOL=15 TO 0 STEP -1:SOUND 0
,O,10,VOL:SOUND 1,O+10,10,VOL
25002 SOUND 2,O+20,10,VOL:SOUND 3,O+30,10,VOL:NEXT VOL:GOSUB 27100:RETURN
26000 ? "}":SETCOLOR Q2,13,Q6:SETCOLOR Q4,13,Q6:SETCOLOR Q1,13,13:POKE 752,1:RET
URN
27000 FOR Z=Q1 TO 200:NEXT Z:RETURN
27100 GRAPHICS 3+16:COLOR 5:PLOT Q10,Q1:DRAWTO Q12,Q1:DRAWTO Q12,Q2:DRAWTO 13,Q2
:DRAWTO 14,Q2:DRAWTO 14,Q1:DRAWTO 16,Q1:DRAWTO 16,Q5:DRAWTO 18,Q5:DRAWTO 18,Q3
27110 DRAWTO 23,Q3:DRAWTO 23,Q5:DRAWTO 25,Q5:DRAWTO 25,Q1:DRAWTO 27,Q1:DRAWTO 27
,Q2:DRAWTO 29,Q2:DRAWTO 29,Q1:DRAWTO 31,Q1:DRAWTO 31,20:DRAWTO Q10,20
27120 DRAWTO Q10,Q1:PLOT 18,19:DRAWTO 18,15:DRAWTO 19,14:DRAWTO 20,13:DRAWTO 21,
13:DRAWTO 22,14:DRAWTO 23,15:DRAWTO 23,19
27130 GOSUB 27000:RETURN
28000 SETCOLOR Q2,Q9,Q1:SETCOLOR Q4,Q9,Q1
28010 SOUND Q0,81,Q10,Q10:GOSUB 28100:SOUND Q1,64,Q10,Q10:GOSUB 28100:SOUND Q2,5
3,Q10,Q10:GOSUB 28100:SOUND Q3,40,Q10,Q10
28020 FOR TIME=Q1 TO 100:NEXT TIME:FOR ZZ=Q0 TO Q3:SOUND ZZ,Q0,Q0,Q0:NEXT ZZ:RET
URN
28100 FOR TIME=Q1 TO Q10:NEXT TIME:RETURN
29000 FOR O=15 TO Q0 STEP -1:SOUND Q0,Q10,Q8,O:NEXT O:SOUND Q0,Q0,Q0,Q0:GOSUB 26
000:RETURN
30000 FOR O=15 TO Q0 STEP -0.2:SOUND Q0,O,Q8,O:NEXT O:GOSUB 26000:RETURN
30100 GOSUB 27000:GOSUB 26000:T=Q0
30103 ? "You made it to the enchanted castle":I=INT(RND(Q1)*21)*Q100:J=INT(RND(Q
1)*Q9):A(X1,Y1)=A(X1,Y1)-Q10000
30110 GOSUB 25000:? "You found ";I;" treasure points there":TL=TL+I:IF J<>Q7 OR
M1=Q7 THEN 30125
30120 ? "You also found a mirror which will     kill any Basilisks you meet":M1=Q
7
30125 J=INT(RND(Q1)*20):IF J=Q2 THEN C=Q2*C
30130 IF J=Q2 THEN ? "You also found an enchanted sword which doubles your stren
gth."
30140 FOR I=Q1 TO CS-Q1:IF C(I)<>X1 THEN NEXT I:GOTO 30145
```

```
30141 FOR J=Q1 TO CS-Q1:C(J)=C(J+Q1):D(J)=D(J+Q1):NEXT J
30144 NEXT I
30145 CS=CS-Q1:IF CS=Q0 THEN ? "You found the last castle!"
30150 RETURN
31000 FOR O=Q1 TO Q5:FOR P=15 TO Q0 STEP -3:SOUND Q0,15,Q2,P:SOUND Q0,20,Q2,P:NE
XT P:NEXT O:SOUND Q0,Q0,Q0,Q0
31001 RETURN
32000 I=INT(RND(Q1)*Q11+Q1)
32001 ON I GOTO 32010,32020,32015,32030,32040,32050,32060,32070,32080,32090,3209
5
32010 POSITION 0,12:? "You stepped into a time warp and lost":? "7 days":DA=DA+Q
7:RETURN
32015 I=INT(RND(Q1)*Q10+Q1):J=DA:DA=DA-I:IF DA<0.1 THEN DA=0.1:I=J-DA
32017 POSITION 0,12:? "You stepped into a time warp and gained":? I;" days":RETU
RN
32020 IF C>=D THEN RETURN
32023 GOSUB 19000:? "You met an elf who gave you a magic   drink that gave your
strength back.":C=D:RETURN
32030 IF V+R+S=V1+R1+S1 THEN RETURN
32033 GOSUB 19000:? "You ran into a wizard who gave you a  potion that restored
all your magic.":V=V1:R=R1:S=S1
32035 RETURN
32040 IF TL<Q2 THEN RETURN
32043 GOSUB 21000:? "You fell into some quicksand. You lost half of your treasur
e":TL=INT(TL/Q2):RETURN
32050 GOSUB 22000:? "You ran into some thick underbrush and used up half your st
rength":C=INT(C/Q2):RETURN
32060 I=INT(RND(Q1)*50+Q1):POSITION Q1,Q12:? "You found ";I;" coins lying on the
 ground":TL=TL+I:RETURN
32070 IF M1<>Q7 THEN RETURN
32073 GOSUB 23000:? "You tripped over some roots and broke your mirror":M1=Q0:RE
TURN
32080 POSITION Q2,Q12:? "A hermit told you that there are ";CS;"    castles left
":RETURN
32090 IF V+S+R=0 THEN RETURN
32091 GOSUB 24000:? "You wandered into an area where magic doesn't work.":V=Q0:S
=Q0:R=Q0:RETURN
32095 IF CS=Q0 THEN RETURN
32096 POSITION Q2,Q12:? "You met a hunter who told you of the  legend of a castl
e ";:I=INT(RND(Q1)*CS)+Q1:GOSUB 1303
32097 RETURN
```

Scurry

David Bohlke

In *Scurry* you are presented with a series of tasks to be accomplished within a limited amount of time, with obstacles to be avoided. The tasks consist of X shaped targets that appear on the screen for a brief period of time. You must use the joystick to move the cursor over the target before it disappears. The obstables are blocks on the screen.

You receive 10 points for each target you reach, and lose 2 points for each obstable you hit. Your cursor is continuously moving, so the game is not easy. As your score increases, the cursor moves faster and more obstacles appear.

Scurry was written to demonstrate several of the special abilities of the Atari computer, including the use of joysticks and Player-Missile graphics.

David Bohlke, Lynn Drive, Coggon, IA 52218.

Scurry

The main feature of this demonstration game is the machine language routine stored in Y$ in line numbers 90 to 97. Objects in P/M graphics can be moved horizontally on the screen with animation speed by using the appropriate POKE addresses. However, there is no such provision for the up and down movement of the graphic images to any vertical location. It is necessary to reposition the entire image, one byte at a time, in the P/M graphics list. If you do this using Basic POKEs the speed of the graphics will be unacceptably slow.

One way to speed up this vertical movement is to reposition the image using a machine language routine. The following code will accomplish this task:

162,20		LDX, 20 20 is the number of bytes to be moved.
189,0,0	LOOP	LDA, Source address This will be the location of your P/M image. In this case, the cursor image is permanently stored at the top of the Player 0 display list. Since the first twenty bytes of the list aren't shown on the screen, this seemed like a convenient location.
157,0,0		STA, DESTINATION address This will be the vertical location in the Player O list (and the screen) that the image will be displayed.
202		DEX decrement counter
208,247		BNE, LOOP Move another byte, for a count of twenty in this example.
104		PLA
96		RTS

Line number 95 of the listing sets UY as the machine language addresses for the USR function. In line number 97, the source address for the image (POKE m+5, MB+1) is set at the beginning of the Player O list. The decimal 250 effectively moves this address six bytes below the Player O image. Since a total of 20 bytes are moved, these extra blank bytes before and after the image will erase the previously set image if the vertical change is restricted to plus or minus five bytes. Hence, it is not necessary at each move to clear the entire Player O list.

The POKE M+8, MB+2 instruction in line 97 will place the MSB destination byte in the routine. Since PY = M+7, the LSB byte can be set each time you want to move the image. Now, the only commands needed for vertical movement are POKE PY,SV: Z=USR (UY) where SV is the vertical placement in the Player O list.

If you have several images to move, it will be necessary to POKE changes for both the source and destination addresses in the routine. Or, perhaps it would be better to set up a different machine language routine for each image. Although this machine code may not be appropriate every time you need to use Player/Missile graphics; hopefully you will find some benefit from the description.

Scurry was not designed as a game to give you 'endless' pleasure until 4 a.m. each morning — even though it is fun for a change of pace from time to time. But it will be worthwhile if it helps you to piece together some of the concepts in Player/Missile graphics.

Line description

5-97	initialization
10	Y$(15) holds the machine language routine D(15) holds directions for STICK commands
11-13	colors
20	MB is 12 pages down from the top of memory PB is the beginning of Player/Missile graphics list
25	reset color attract mode, turn off cursor
30-40	initialize P/M graphics
50-52	put cursor image at bottom of Player O area in P/M graphics list
54-56	size and color of Player O
90-97	put machine language routine into Y$
100-270	pre-game initialization
100	P stores points accumulated PK is the increment of 100 counter
110	SH,SV are the cursor horizontal and vertical positions
120-122	prints
150	position target
160-162	directions for STICK function
200-270	T is time, D is direction X,Y are speed increments for the cursor
300-500	main game loop
300	check for collision of cursor and graphics blocs
310	if collision, then decrement points, pick new RND direction, print score
320	decrease time
330-390	check STICK for cursor direction
400	adjust P/M cursor's vertical position
410	plot P/M cursor's horizontal position
420-430	check for hit on target
440	increment points for hit
460	plot new target
480	check if point increment is over 100 multiple
490	increase speed
492	reset time, increment multiple of 100 counter
700-790	set new target
800-820	prompt for next game

Scurry

```
5 REM  SCURRY  by David Bohlke
10 DIM Y$(15),D(15)
11 GRAPHICS 3
12 SETCOLOR 2,14,2:SETCOLOR 4,14,2
13 SETCOLOR 0,8,6:SETCOLOR 1,11,8
20 MB=PEEK(106)-12:POKE 54279,MB:PB=256*MB
25 POKE 77,0:POKE 752,1
26 POKE 656,1:POKE 657,16:PRINT " SCURRY "
```

```
30 POKE 559,46:REM DOUBLE LINE GR.
32 POKE 53277,3:REM ENABLE GRAPHICS
34 POKE 623,8:REM PLYR PRIORITY
40 FOR I=PB+500 TO PB+800:POKE I,0:NEXT I:REM ZERO GR. AREA
50 RESTORE 52:FOR I=PB+512 TO PB+516:READ A:POKE I,A:NEXT I
51 REM PUT PL0 AT BOTTOM OF PL0 AREA
52 DATA 14,17,21,17,14
54 POKE 53256,0:REM SIZE OF PL0
56 POKE 704,0:REM COLOR OF PL0
90 M=ADR(Y$):RESTORE 92:FOR I=1 TO 13:READ A:POKE M+I,A:NEXT I
91 REM MACHINE LANG. ROUTINE
92 DATA 162,20,189,250,0,157,0,0,202,208,247,104,96
95 UY=M+1:REM POKE PY,SV:Z=USR(UY)
96 REM SOURCE,DESTINATION OF ROUTINE
97 POKE M+5,MB+1:POKE M+8,MB+2:PY=M+7
100 P=0:PK=0
110 SH=125:SV=60
120 POKE 656,3:POKE 657,1:PRINT "SCORE ";
122 POKE 656,3:POKE 657,30:PRINT "TIME";
150 GOSUB 700
160 RESTORE 162:FOR I=5 TO 15:READ A:D(I)=A:NEXT I
162 DATA 2,1,2,0,3,4,4,0,3,1,0
200 T=500
210 D=INT(RND(0)*4)+1
270 X=3:Y=3
299 REM MAIN PLAY LOOP
300 C=PEEK(53252):POKE 53278,0
310 IF C=2 OR C=1 THEN P=P-2:SOUND 0,50*C,10,14:D=INT(RND(0)*4)+1:POKE
    657,7:PRINT P;" ";
320 T=T-1:POKE 656,3:POKE 657,35:PRINT INT(T);" ";:IF T<1 THEN 800
330 S=STICK(0):IF S=15 THEN 350
340 D=D(S)
350 GOTO D*10+350
360 SV=SV-Y:IF SV<10 THEN SV=10:D=3
362 GOTO 400
370 SH=SH+X:IF SH>200 THEN SH=200:D=4
372 GOTO 400
380 SV=SV+Y:IF SV>90 THEN SV=90:D=1
382 GOTO 400
390 SH=SH-X:IF SH<50 THEN SH=50:D=2
400 POKE PY,SV:Z=USR(UY)
405 SOUND 0,0,0,0
410 POKE 53248,SH
420 C=PEEK(53260)
430 IF C<>2 THEN 300
440 P=P+10:POKE 656,3:POKE 657,7:PRINT P;" ";
450 SOUND 1,222,6,12:SOUND 2,100,2,14
460 GOSUB 700
462 POKE 53278,0
464 SOUND 1,0,0,0:SOUND 2,0,0,0
480 IF INT(P/100)<PK THEN 300
490 X=X+0.5:Y=Y+0.5:IF Y>5 THEN Y=5:X=5
492 T=500:PK=PK+1
500 GOTO 300
```

```
700 TH=RND(0)*140+55:REM TARGET HORIZ
704 REM MOVE OLD TGT. OFF OF SCREEN
705 POKE 53249,10
710 FOR I=TV+640+PB TO TV+PB+646:POKE I,0:NEXT I:REM CLEAR OLD PL1 TARGET
720 CL=INT(RND(0)*15)*16+8:POKE 705,CL:REM COLOR OF PL1
730 POKE 53257,0:REM SIZE OF PL1
750 TV=RND(0)*70+20:REM TARGET VERT.
760 RESTORE 762:FOR I=TV+640+PB TO TV+PB+646:READ A:POKE I,A:NEXT I
761 REM PUT TARGET INTO PL1 AREA
762 DATA 65,34,28,28,28,34,65
770 POKE 53249,TH:REM PLOT NEW TGT.
781 C=INT(RND(0)*2)+1:IF P>200 AND RND(0)<0.8 THEN 790
782 COLOR C:PLOT RND(0)*39,RND(0)*19
790 RETURN
800 PRINT :PRINT "Press START for next game ?";
810 IF PEEK(53279)=6 THEN 11
820 SOUND 0,RND(0)*200,10,2:GOTO 810
```

Collision

David Bohlke

Now is the time for iron nerves and instant reflexes. Your nimble sports car will soon by flying down the dreadful ribbon pavement known as the "Serpent of Oblivion." There's no chance of turning around or slowing down — you must forge ahead until the inevitable collision. But you will succeed if you can survive longer and score more points than any of your opponents.

To begin your challenge of the Serpent, plug a joystick into Slot #1. One to four players can compete in each game. Every player will have three turns to navigate the swooning roadway. Before play starts, you must also select a skill sevel (1-5). This will determine the width of the track.

During play, your car will be at the top of the screen. The curving road will move from the bottom of the screen to the top. To avoid a collision, you must maneuver your car to remain on the roadway while also avoiding any obstructions on the road. Steering your car is a matter of pushing the joystick to the left or right. The 'wheel' is very sensitive, so it may take some practice to get the feel of the car. The longer you push the stick (left or right) the faster your car will veer in that direction.

Scoring is determined by the skill level selected and the length of time you survive before the certain collision. At the end of each player's turn, a score card showing each drivers' score for every turn will be displayed. The higher your score, the better your driving ability on the Serpent. At this time, the player who is to drive next will be prompted to press the fire button which will initiate your turn.

The program listing for collision should be fairly easy to decipher. Lines 5-30 are the initialization. N$ will hold the players' names, S(4,3) contains each player's score for each round, and R$(20) holds the graphics string for the roadway. Lines 50-172 are the prompts for the beginning of the game options.

The main game loop is in lines 180-410. Your car is displayed using Player/Missile graphics. The car position is printed in line 370; and a collision is checked for in line 380. Line 320 prints the road obstacles and adjusts a counter to increase their frequency as your turn progresses.

In the subroutine at 500-550, the sounds and colors are changed after a collision. The subroutine at 600-699 displays the score card. Lines 900-960 set up the Player/Missile graphics and the routine at 970 formats the string to print the roadway.

David Bohlke, Lynn Drive, Coggon, IA 52218.

Collision

```
5 REM  COLLISION   by David Bohlke
10 GRAPHICS 0:POKE 752,1
15 DIM B$(6),N$(32),N(4),S(4,3),R$(20)
18 FOR I=0 TO 4:FOR J=0 TO 3:S(I,J)=0:NEXT J:
   NEXT I
20 SETCOLOR 2,13,4:SETCOLOR 4,4,2
```

```
30 H=120:GOSUB 900:LD=1
40 PRINT "}":POSITION 13,3:PRINT " COLLISION "
50 POSITION 3,6:PRINT "How many players (1-4) ";:
   INPUT NP:IF NP<1 OR NP>4 THEN 4 0
55 FOR I=1 TO NP:PRINT :PRINT "ENTER name of player # ";I;"  ";
60 INPUT B$:N(I)=LEN(B$)+N(I-1):N$(N(I-1)+1,N(I))=B$:NEXT I
100 PRINT "}":POSITION 13,3:PRINT " COLLISION "
110 POSITION 2,10:PRINT "PUSH STICK to enter SKILL LEVEL,":PRINT
    "then press the FIRE button . . ."
120 POSITION 2,16:PRINT " 1  is easiest,  5  is hardest ?"
150 POSITION 35,16:PRINT LD;
152 FOR I=1 TO 50:NEXT I
155 IF STRIG(0)=0 THEN 172
160 IF STICK(0)=15 THEN 150
165 LD=LD+1:IF LD=6 THEN LD=1
170 GOTO 150
172 L=13-LD:GOSUB 970
174 PRINT :PRINT :PRINT N$(1,N(1));" --- press FIRE for your turn ?";
175 FOR I=1 TO 333:NEXT I
176 IF STRIG(0)=1 THEN 176
180 FOR RD=1 TO 3:FOR PL=1 TO NP
190 C1=15:PRINT "}"
200 A=12:B=0.5:C=0:P=0:K=0
210 POKE 77,0:S=0
220 POKE 53278,0
250 FOR I=0 TO 23:POSITION 12,I:PRINT R$:NEXT I
300 POSITION A,23:PRINT R$
305 SOUND 0,100-ABS(S)*10,4,ABS(S)*2+2
310 A=A+B:IF A>22 OR A<1 OR RND(0)<0.01 THEN B=-B
320 C=C+1:IF C=C1 THEN POSITION A+RND(0)*(L-3)+3,22:PRINT
     ";:C=0:P=P+LD*2:IF INT (P/100)>K THEN K=K+1:C1=C1-1
340 P=P+1:POSITION 1,0:PRINT P;
350 IF STICK(0)<8 THEN S=S+1:GOTO 370
360 IF STICK(0)<12 THEN S=S-1
370 H=H+S:POKE 53248,H
380 IF PEEK(53252)<>0 THEN GOSUB 500:GOTO 400
390 GOTO 300
400 S(PL,RD)=P:GOSUB 600
405 NEXT PL
410 NEXT RD
420 END
500 SOUND 0,0,0,0
510 FOR I=1 TO 5:SETCOLOR 4,RND(0)*16,
    RND(0)*16:SETCOLOR 2,RND(0)*16,RND(0)*16
520 FOR J=1 TO 5:SOUND 0,RND(0)*30,4,8:SOUND 1,RND(0)*I*10,8,8:NEXT J
522 NEXT I:SOUND 0,0,0,0:SOUND 1,0,0,0
540 H=120:POKE 53248,H
550 RETURN
600 PRINT "}":POSITION 3,1:PRINT " COLLISION "
605 SETCOLOR 2,1,2:SETCOLOR 4,11,4
610 POSITION 3,4:PRINT "Round:   One    Two    Three   TOTAL"
620 FOR I=1 TO NP:POSITION 1,I*4+3:PRINT N$(N(I-1)+1,N(I));
622 FOR J=0 TO 39:POSITION J,I*4+5:PRINT " ";:NEXT J:NEXT I
```

```
630 FOR I=1 TO NP:T=0
635 FOR J=1 TO RD:T=T+S(I,J)
640 POSITION J*7+5,I*4+3:PRINT S(I,J);:POSITION 34,I*4+3:PRINT
    T;:NEXT J:NEXT I
680 P=PL+1:IF P>NP THEN P=1:IF RD=3 THEN 697
690 POSITION 1,23:PRINT N$(N(P-1)+1,N(P));" ---
    Press FIRE for your turn ?";
692 IF STRIG(0)=1 THEN 690
693 SETCOLOR 2,13,4:SETCOLOR 4,4,2
695 RETURN
697 POSITION 1,23:PRINT "Press FIRE for next game ??????";
698 IF STRIG(0)=1 THEN 697
699 RUN
900 MB=PEEK(106)-8:POKE 54279,MB:PB=256*MB
910 POKE 559,46:POKE 53277,3
920 FOR I=PB+512 TO PB+640:POKE I,0:NEXT I
930 POKE 53248,H:POKE 704,122:POKE 53256,0
940 FOR I=PB+530 TO PB+536:READ A:POKE I,A:NEXT I
950 DATA 51,63,30,12,12,63,51
960 RETURN
970 R$=""
972 FOR I=1 TO L:R$(LEN(R$)+1)=" ":NEXT I
974 R$(LEN(R$)+1)="":RETURN
```

Air Defense

David Bohlke

There's a spy in the sky! How long are you going to allow this super-snooper to fly in your airspace? Sure, they can hide behind the clouds or swoop below your mountain range, but you should be able to stop them with your deadly sonic cannon. Be warned — these sleuths are infinitely persistant and you can't possibly get them all. But, with an accurate eye, you should be able to maintain your gunners rank.

To begin play, plug a joystick into Slot #1. At the start, you'll also be able to select the speed of the spy craft. This is on a scale of one to five, with one being the slowest. It will take a little practice before you can advance to the faster games.

David Bohlke, Lynn Drive, Coggon, IA 52218.

During the game, the spy planes will move horizontally across the screen. Use your joystick to position your cannon and fire at the snooper. Besides leading the plane according to its speed, you'll also have to adjust the height of burst for your sonic cannon. The current altitude setting (1-9) will be displayed on the screen. To move the cannon horizontally, push the stick to the left or right. Adjusting the altitude is accomplished by pushing the stick up or down. When you're set to shoot, press the fire button.

It will take a few practice rounds for you to get a feel for the various altitudes and speeds. You can use the clouds and mountain landscape as a gauge for altitude. Usually, only one specific altitude setting will accomplish a hit on the plane. Faster and/or higher planes will require a little more lead horizontally and the aircrafts are most vulnerable on their lower tail sections.

Scoring is done in a progressive manner. You begin with fifteen rounds for the sonic blaster and for every increment of 500 points that you accumulate your ammunition supply will be replenished to 15. The game will continue as long as you have ammunition remaining. For each hit, you can score from 40 to 80 points, depending on the altitude of the plane. This score and your total score will be displayed on the lower left of the screen. Points will be deducted from your score everytime you miss (minus 20) and when a plane escapes off the edge of the screen (minus 50). If you can consistantly score over 2000 points, then you should select a faster speed for a higher skill level.

Air Defense illustrates an extensive use of Player/Missile graphics. The planes (left and right) are PLO and are formatted in lines 700 to 750. The PL1 mode is used both for the cannon smoke (800-808) and for the sonic blast (860-870). Your cannon is PL2 and is set in lines 820-828. PL3 represents the fireball when you score a hit. This routine at 830-850 increases in size as the fireball expands. The terrain and clouds are set under Graphics 5 in lines 880-899.

The basic game initialization is in lines 3-140. Line 92 has DATA statements used in reading the STICK commands for the horizontal and altitude increments. The main game loop is in lines 150 to 490. Lines 150-195 set the plane and adjust the prints; and lines 200-220 check for the plane being off the screen. All of the 300's read and interpret the STICK commands. The 400's control the gun firing and check for a hit on the plane. Line 460 checks the proximity blast and may be adjusted for easier accuracy. When there is a hit, the program branches to line 500 and then returns to the main game loop. Finally, lines 900-970 print the score and test for the end of the game.

Air Defense

```
3 REM  AIR DEFENSE
4 REM by David Bohlke, Coggon, IA
5 GRAPHICS 5:POKE 752,1
10 SETCOLOR 4,9,4:REM SKY
11 SETCOLOR 2,12,4:REM EARTH
12 SETCOLOR 0,0,10:REM CLOUDS
20 A=PEEK(106)-12:POKE 54279,A:PMBASE=256*A
22 DIM A(15),H(15)
30 POKE 559,46:REM DBL LINE GR
32 POKE 53277,3:REM ENABLE GR.
34 POKE 623,8:REM FIELD PRIORITY
40 POKE 53256,0:REM SIZE PL0
42 POKE 53257,1:REM SIZE PL1
45 POKE 53258,0:REM SIZE PL2
46 POKE 706,2:REM COLOR PL2
90 RESTORE 92:FOR I=1 TO 15:READ A,H:A(I)=A:H(I)=H:NEXT I
92 DATA 0,0,0,0,0,0,0,0,-1,1,1,1,0,1,0,0,-1,-1,1,
   -1,0,-1,0,0,-1,0,1,0,0,0
94 PS=1:HF=PS+3:PLN=0:RD=15:RC=0:PP=20
100 GOSUB 880:GOSUB 890
105 PRINT ," AIR DEFENSE ":PRINT
106 PRINT " SELECT  SPEED, then press  START ":PS=1
107 POKE 656,2:POKE 657,37:PRINT PS;
108 IF PEEK(53279)=6 THEN 113
110 IF PEEK(53279)=5 THEN PS=PS+1:IF PS>5 THEN PS=1
111 FOR J=1 TO 50:NEXT J
112 SOUND 0,200*RND(0),10,2:GOTO 107
113 G=PS:PS=PS*0.5+0.5:HF=PS+3:? :? :? :?
114 SOUND 0,0,0,0
115 GOSUB 800:GOSUB 820
120 GH=125:POKE 53250,GH:GA=5
130 POKE 656,2:POKE 657,2:PRINT "Game  ";G;
150 GOSUB 700:AK=10-INT((PA-12)/5)
170 FOR I=PMBASE+640 TO PMBASE+768:POKE I,0:NEXT I
175 POKE 77,0:POKE 53259,0
180 I=INT(PT/500):IF I>RC THEN RC=I:RD=15:PP=PP+5
185 POKE 656,2:POKE 657,30:PRINT "Rnds  ";RD;" ";
190 IF RD<1 THEN 950
195 PLN=PLN+1:GOSUB 900
200 PH=PH+DR*PS
210 IF DR=1 AND PH>PE THEN PT=PT-50:RD=RD-1:GOTO 150
212 IF DR=-1 AND PH<PE THEN PT=PT-50:RD=RD-1:GOTO 150
```

```
220 POKE 53248,PH
250 IF PEEK(53279)=6 THEN RUN
300 S=STICK(0)
302 SOUND 3,30,8,INT(PA/20)+1
305 GH=GH+H(S)*HF:IF GH<50 THEN GH=50
306 IF GH>200 THEN GH=200
310 GA=GA+A(S):IF GA>9 THEN GA=9
312 IF GA<1 THEN GA=1
320 POKE 53250,GH
349 I=INT((GH-50)/42.5)
350 POKE 656,3:POKE 657,17:PRINT "ALT ";GA;
360 SOUND 3,30,4,INT(PA/20)+2
400 IF STRIG(0)<>0 THEN 200
405 GOSUB 801:RD=RD-1
406 POKE 656,2:POKE 657,30:PRINT "Rnds  ";RD;" ";
420 FOR I=1 TO AK:PH=PH+DR*PS
421 POKE 53249,GH-8+RND(0)*9
422 FOR J=1 TO 15:NEXT J
424 SOUND 2,60,4,15
426 POKE 53248,PH:NEXT I
427 SOUND 2,0,0,0
429 POKE 705,34
430 FOR I=PMBASE+640+CH TO PMBASE+645+CH:POKE I,0:NEXT I
450 GOSUB 860:POKE 53249,GH-4
455 FOR I=1 TO 30:SOUND 2,I+20,4,14:SOUND
     3,60,8,14:NEXT I:SOUND 2,0,0,0:SOUND 3,0,0,0
460 IF ABS(AB-PA)<3 AND ABS(GH-PH)<5 THEN PT=PT+100-PA:GOTO 500
470 PT=PT-PP:GOSUB 900
480 FOR I=PMBASE+640+AB TO PMBASE+647+AB:POKE I,0:NEXT I
482 POKE 53249,20
485 IF RD<1 THEN 950
490 GOTO 200
500 POKE 53249,20:GOSUB 900
505 SOUND 3,0,0,0
510 GOSUB 831:POKE 53248,20
540 FOR I=PMBASE+896 TO PMBASE+1024:POKE I,0:NEXT I
550 POKE 53251,20
560 GOTO 150
700 FOR I=PMBASE+512 TO PMBASE+640:POKE I,0:NEXT I
702 PA=INT(RND(0)*40)+20:CL=INT(RND(0)*16)*16+2
710 IF RND(0)<0.5 THEN 720
712 PH=35:PE=213:DR=1:RESTORE 718
717 GOTO 730
718 DATA 56,156,206,255,14,28,56
720 PH=213:PE=35:DR=-1:RESTORE 728
728 DATA 28,57,115,255,112,56,28
730 FOR I=PMBASE+512+PA TO PMBASE+518+PA:READ A:POKE I,A:NEXT I
732 POKE 53256,0:REM SIZE PL0
734 POKE 704,CL:REM COLOR PL0
750 RETURN
800 FOR I=PMBASE+640 TO PMBASE+768:POKE I,0:NEXT I
801 RESTORE 808:CH=94:POKE 705,12
802 FOR I=PMBASE+640+CH TO PMBASE+645+CH:READ A:POKE I,A:NEXT I
805 RETURN
```

```
808 DATA 24,36,90,165,195,60
820 FOR I=PMBASE+768 TO PMBASE+896:POKE I,0:NEXT I
821 RESTORE 828:PL=100
822 FOR I=PMBASE+768+PL TO PMBASE+779+PL:READ A:POKE I,A:NEXT I
825 RETURN
828 DATA 24,24,24,24,24,24,60,60,90,90,219,255
830 FOR I=PMBASE+896 TO PMBASE+1024:POKE I,0:NEXT I
831 RESTORE 840
832 FOR I=PMBASE+896+PA TO PMBASE+906+PA:READ A:POKE I,
    A:NEXT I:POKE 707,66
834 POKE 53259,0:POKE 53251,PH
840 DATA 74,149,72,34,145,40,66,169,68,146
845 FOR I=1 TO 10:SOUND 0,RND(0)*30+30,4,4:
    SOUND 1,RND(0)*50+150,4,4:NEXT I
846 POKE 53259,1:POKE 707,68
847 FOR I=1 TO 10:SOUND 0,RND(0)*30+30,4,8:SOUND
    1,RND(0)*50+150,4,8:NEXT I
848 POKE 53251,PH-9:POKE 53259,3:POKE 707,72
849 FOR I=1 TO 10:SOUND 0,RND(0)*30+30,4,14:SOUND
    1,RND(0)*50+150,4,14:NEXT I
850 SOUND 0,0,0,0:SOUND 1,0,0,0:RETURN
860 FOR I=PMBASE+640+AB TO PMBASE+647+AB:POKE I,0:NEXT I
861 RESTORE 866:AB=(10-GA)*5+15
862 FOR I=PMBASE+640+AB TO PMBASE+647+AB:READ A:POKE I,A:NEXT I
866 DATA 129,66,20,43,212,40,66,129
870 RETURN
880 COLOR 3:D=35:E=1
882 FOR I=0 TO 79:PLOT I,D:DRAWTO I,40
884 IF RND(0)<0.2 THEN E=-E
886 D=D+E:IF D>40 THEN D=40
887 IF D<20 THEN D=20
888 NEXT I:RETURN
890 COLOR 1:X=10+RND(0)*20:X1=X:Y=RND(0)*20:Z=Y:GOSUB 893
891 X=45+RND(0)*20:Y=RND(0)*20:X1=X:IF ABS(Y-Z)<8 THEN 891
892 GOSUB 893:RETURN
893 D=8+RND(0)*8
894 FOR I=1 TO D
895 L=RND(0)*4*-1:R=RND(0)*4:IF I>D/2 THEN L=-L/2:R=-R/2
896 X=X+L:X1=X1+R:IF X<0 THEN X=0
897 IF X1>79 THEN X1=79
898 PLOT X,Y:DRAWTO X1,Y:Y=Y+1:NEXT I
899 RETURN
900 IF PT<0 THEN PT=0
905 POKE 656,3:POKE 657,2:PRINT "Score ";PT;" ";
910 POKE 656,3:POKE 657,30:PRINT "Plane ";PLN;
920 RETURN
950 POKE 656,0:POKE 657,2:PRINT "PRESS  START  FOR NEXT GAME ??";
955 POKE 53248,20
960 IF PEEK(53279)=6 THEN RUN
970 SOUND 0,RND(0)*250,10,2:GOTO 960
```

Appendix

Atari Memory Addresses

Decimal	Hexadecimal	Label	Comment
2,3	ØØØ2 - ØØØ3	CASINI	cassette boot completed vector
6	ØØØ6	TRAMSZ	End of RAM test temporary storage
8	ØØØ8	WARMST	Warm start flag (-1 = true)
1Ø,11	ØØØA - ØØØB	DOSVEC	no cartridge control vector (start)
12,13	ØØØC - ØØØD	DOSINI	disk boot completed vector
14,15	ØØØE - ØØØF	APPMHI	Highest location used by Basic
16	ØØ1Ø	POKMSK	ANTIC interrupt register storage
17	ØØ11	BRKKEY	Break key flag (-1 = false)
18-2Ø	ØØ12 - ØØ14	RTCLOK	TV Frame counter
49-52	ØØ31 - ØØ34		Floppy disk serial bus device addresses
64	ØØ4Ø		printer serial bus device address
65	ØØ41	SOUNDR	Sound I/O flag (Ø=quiet)
74	ØØ4A	CKEY	Cassette boot request flag
75	ØØ4B	ATRACT	Attract mode flag (>128 = attract)
8Ø	ØØ5Ø		modem serial bus device address
82	ØØ52	LMARGN	Left screen margin (default = 2)
83	ØØ53	RMARGN	Right screen margin (default = 37)
84	ØØ54	ROWCRS	Current cursor row (Ø-39)
85-86	ØØ55 - ØØ56	COLCRS	Current cursor column (Ø-23)
9Ø	ØØ5A	OLDROW	Previous cursor row (Ø-39)
91-92	ØØ5B - ØØ5C	OLDCOL	Previous cursor column (Ø-23)
93	ØØ5D	OLDCHR	Data under cursor
96	ØØ6Ø	NEWROW	Cursor row to which DRAWTO goes
97-98	ØØ61 - ØØ62	NEWCOL	Cursor column to which DRAWTO goes
1Ø6	ØØ6A	RAMTOP	Top of memory (Page number)

Decimal	Hexadecimal	Label	Comment
	BASIC area		
128-9	ØØ8Ø - ØØ81	LOMEM	Basic low memory pointer
128-9	ØØ8Ø - ØØ81	OUTBUFF	Syntax output buffer
13Ø-1	ØØ82 - ØØ83	VNTP	Variable name table address
132-3	ØØ84 - ØØ85	VNTD	End of variable name table
134-5	ØØ86 - ØØ87	VNTP	Variable values table
136-7	ØØ88 - ØØ89	STMTAB	Statement table
138-9	ØØ8A - ØØ8B	STMCUR	Immediate statement
14Ø-1	ØØ8C - ØØ8D	STARD	String array table
142-3	ØØ8E - ØØ8F	RUNSTK	Run time stack
144-5	ØØ9Ø - ØØ91	MEMTOP	Basic top of memory pointer
186-7	ØØBA - ØØBB	STOPLN	Line number for TRAP or STOP
195	ØØC3	ERRSAV	Error number
2Ø1	ØØC9	PTABW	Print tab width (default = 1Ø)
212-3	ØØD4 - ØØD5	FRØ	Value returned to Basic from USR
212-8	ØØD4 - ØØDA	FRØ	6 byte floating point handler
224-9	ØØEØ - ØØE5	FR1	6 byte floating point handler
242	ØØF2	CIX	index offset used with INBUFF (ØØF3)
243-4	ØØF3 - ØØF4	INBUFF	pointer to ASCII text buffer
251	ØØFB	RADFLG	Radian/degree flag (Ø RAD - 6 DEG)
252-3	ØØFC - ØØFD	FLPTR	pointer to a floating point number

Decimal	Hexadecimal	Label	Comment
(Mostly Vectors and Shadow Registers)			
256-511	Ø1ØØ - Ø1FF	STACK	65Ø2 stack area
512-3	Ø2ØØ - Ø2Ø1	VDSLST	RTI vector (E7B3 = ignore interrupt)
514-5	Ø2Ø2 - Ø2Ø3	VPRCED	Serial I/O interrupt proceed vector
516-7	Ø2Ø4 - Ø2Ø5	VINTER	Serial I/O interrupt vector
518-9	Ø2Ø6 - Ø2Ø7	VBREAK	65Ø2 Break instruction vector
52Ø-1	Ø2Ø8 - Ø2Ø9	VKEYBD	Key pressed interrupt vector
522-3	Ø2ØA - Ø2ØB	VSERIN	Serial bus input ready vector
524-5	Ø2ØC - Ø2ØD	VSEROR	Serial bus output ready vector
526-7	Ø2ØE - Ø2ØF	VSEROC	Serial bus output complete vector
528-9	Ø21Ø - Ø211	VTIMR1	POKEY timer #1 vector
53Ø-1	Ø212 - Ø213	VTIMR2	POKEY timer #2 vector
532-3	Ø214 - Ø215	VTIMR4	timer vector
534-5	Ø216 - Ø217	VIMIRQ	immediate IRQ global RAM vector
536-7	Ø218 - Ø219	CDTMV1	SIO timeout timer
538-9	Ø21A - Ø21B	CDTMV2	timer #2
54Ø-1	Ø21C - Ø21D	CDTMV3	timer #3
542-3	Ø21E - Ø21F	CDTMV4	timer #4
544-5	Ø22Ø - Ø221	CDTMV5	timer #5
546-7	Ø222 - Ø223	VVBLK1	vertical blank RAM vector
548-9	Ø224 - Ø225	VVBLKD	vertical blank deferred vector
55Ø-1	Ø226 - Ø227	CDTMA1	vector for CDTMV1 at Ø218
552-3	Ø228 - Ø229	CDTMA2	vector for CDTMV2 at Ø21A
554	Ø22A	CDTMF3	flag for CDTMV3 at Ø21C
556	Ø22C	CDTMF4	flag for CDTMV4 at Ø21E
558	Ø22E	CDTMF5	flag for CDTMV5 at Ø22Ø
559	Ø22F	SDMCTL	data from ANTIC DMACTL (D4Ø3)
56Ø	Ø23Ø	SDLSTL	data from ANTIC DLISTL (D4Ø2)
561	Ø231	SDLSTH	data from ANTIC DLISTH (D4Ø3)
564	Ø234	LPENH	light pen data from PENH (D4ØC)
565	Ø235	LPENV	light pen data from PENV (D4ØD)
576	Ø24Ø	DFLAGS	disk boot file flag
58Ø	Ø244	COLDST	cold start flag

Decimal	Hexadecimal	Label	Comment
623	Ø26F	GPRIOR	data from CTIA PRIOR (DØ1B)
624	Ø27Ø	PADDLØ	Pot Ø data from POTØ (D2ØØ)
625	Ø271	PADDL1	Pot 1 data from POT1 (D2Ø1)
626	Ø272	PADDL2	Pot 2 data from POT2 (D2Ø2)
627	Ø273	PADDL3	Pot 3 data from POT3 (D2Ø3)
628	Ø274	PADDL4	Pot 4 data from POT4 (D2Ø4)
629	Ø275	PADDL5	Pot 5 data from POT5 (D2Ø5)
63Ø	Ø276	PADDL6	Pot 6 data from POT6 (D2Ø6)
631	Ø277	PADDL7	Pot 7 data from POT7 (D2Ø7)
632	Ø278	STICKØ	joystick Ø data (PORTA D3ØØ)
633	Ø279	STICK1	joystick 1 data (PORTA D3ØØ)
634	Ø27A	STICK2	joystick 2 data (PORTB D3Ø1)
635	Ø27B	STICK3	joystick 3 data (PORTB D3Ø1)
644	Ø284	STRIGØ	joystick trigger data (TRIGØ DØØ1)
645	Ø285	STRIG1	joystick trigger data (TRIG1 DØØ2)
646	Ø286	STRIG2	joystick trigger data (TRIG2 DØØ3)

647	Ø287	STRIG3	joystick trigger data (TRIG3 DØØ4)
656	Ø29Ø	TXTROW	text window cursor row
657-8	Ø291 - Ø292	TXTCOL	text window cursor column
7Ø4	Ø2CØ	PCOLRØ	data from CTIA COLPMØ (DØ12)
7Ø5	Ø2C1	PCOLR1	data from CTIA COLPM1 (DØ13)
7Ø6	Ø2C2	PCOLR2	data from CTIA COLPM2 (DØ14)
7Ø7	Ø2C3	PCOLR3	data from CTIA COLPM3 (DØ15)
7Ø8	Ø2C4	COLORØ	data from CTIA COLPFØ (DØ16)
7Ø9	Ø2C5	COLOR1	data from CTIA COLPF1 (DØ17)
71Ø	Ø2C6	COLOR2	data from CTIA COLPF2 (DØ18)
712	Ø2C7	COLOR3	data from CTIA COLPF3 (DØ19)
713	Ø2C8	COLOR4	data from CTIA COLBK (DØ1A)
736-7	Ø2EØ - Ø2E1	RUNAD	execution address after LOAD
741	Ø2E5	MEMTOP	Top of free RAM (before screen)
743	Ø2E7	MEMLO	Start of free RAM (after BOOT area)
752	Ø2FØ	CRSINH	Cursor inhibit flag
755	Ø2F3	CHACT	Character data (from CHACTL D4Ø1)
756	Ø2F4	CHBAS	Character base address (CHBASE D4Ø9)
763	Ø2FB	ATACHR	Atari Character and color for line
764	Ø2FC	CH	Character read from POKEY
765	Ø2FD	FILDAT	Color for XIO 18 fill
766	Ø2FE	DSPFLG	Display flag
767	Ø2FF	SSFLAG	Start/stop flag (Break)

Ø3ØØ - Ø3ØB Device Control Block for Disk I/O
(set up and JMP to DSKINV (E453))

Decimal	Hexadecimal	Label	Comment
768	Ø3ØØ	DDEVIC	Serial bus ID for disk drive
769	Ø3Ø1	DUNIT	Disk drive number (1-4)
77Ø	Ø3Ø2	DCOMD	Disk command
771	Ø3Ø3	DSTATS	Disk status byte
772	Ø3Ø4	DBUFLO	Disk buffer address (low byte)
773	Ø3Ø5	DBUFHI	Disk buffer address (high byte)
774	Ø3Ø6	DTIMLO	Disk timeout value (seconds)
776	Ø3Ø8	DBYTLO	Disk I/O Byte counter (low byte)
777	Ø3Ø9	DBYTHI	Disk I/O Byte counter (high byte)
778	Ø3ØA	DAUX1	Disk sector number (low byte)
779	Ø3ØB	DAUX2	Disk sector number (high byte)
794	Ø31A	HATABS	Device handler table
832	Ø34Ø	ICHID	Input control handler identification
833	Ø341	ICDNØ	Input control device number
834	Ø342	ICCOM	Input control command byte
835	Ø343	ICSTA	Input control status byte
836	Ø344	ICBAL	Input control buffer address (low)
837	Ø345	ICBAH	Input control buffer address (high)
838	Ø346	ICPTL	Input control pointer (low)
839	Ø347	ICPTH	Input control pointer (high)
84Ø	Ø348	ICBLL	Input control buffer length (low)
841	Ø349	ICBLH	Input control buffer length high
842	Ø34A	ICAX1	Input control auxiliary 1
843	Ø34B	ICAX1	Input control auxiliary 2

14Ø8-	Ø58Ø - Ø5xx	LBUFF	floating point result buffer
1536-1791	Ø6ØØ - Ø6FF		reserved for cartridge when cartridge used
	Ø7ØØ - 12FF	File Management System RAM	
1792	Ø7ØØ	BOOT	flag (DOS only)
1799	Ø7Ø7	FILES	number of files to be open at once
18ØØ	Ø7Ø8	DRIVES	each bit represents an active drive
18Ø2-3	Ø7ØA - Ø7ØB	SASA	disk buffer address
	13ØØ - 267F	Disk Operating System RAM	
4889	1319	LOAD	DOS load file routine
	268Ø - 2A7F	Disk Input/Output buffers	
4Ø956	9FFC		Cartridge B test (Ø = cartridge)
4Ø958	9FFE		Cartridge B initialization vector
49148	BFFC		Cartridge A test (Ø = cartridge)
4915Ø	BFFE		Cartridge A initialization vector

CTIA Chip (DØØØ - DØ1F)

WRITE CTIA addresses

53248	DØØØ	HPOSPØ	Horizontal position of player Ø
53249	DØØ1	HPOSP1	Horizontal position of player 1
5325Ø	DØØ2	HPOSP2	Horizontal position of player 2
53251	DØØ3	HPOSP3	Horizontal position of player 3
53252	DØØ4	HPOSMØ	Horizontal position of missile Ø
53253	DØØ5	HPOSM1	Horizontal position of missile 1
53254	DØØ6	HPOSM2	Horizontal position of missile 2
53255	DØØ7	HPOSM3	Horizontal position of missile 3
53256	DØØ8	SIZEPØ	Size of player Ø
53257	DØØ9	SIZEP1	Size of player 1
53258	DØØA	SIZEP2	Size of player 2
53259	DØØB	SIZEP3	Size of player 3
5326Ø	DØØC	SIZEM	Size of all missiles
53261	DØØD	GRAFPØ	Graphics for player Ø
53262	DØØE	GRAFP1	Graphics for player 1
53263	DØØF	GRAFP2	Graphics for player 2
53264	DØ1Ø	GRAFP3	Graphics for player 3
53265	DØ11	GRAFM	Graphics for missiles
53266	DØ12	COLPMØ	Color of player and missile Ø
53267	DØ13	COLPM1	Color of player and missile 1
53268	DØ14	COLPM2	Color of player and missile 2
53269	DØ15	COLPM3	Color of player and missile 3
5327Ø	DØ16	COLPFØ	Color of playfield Ø
53271	DØ17	COLPF1	Color of playfield 1
53272	DØ18	COLPF2	Color of playfield 2
53273	DØ19	COLPF3	Color of playfield 3
53274	DØ1A	COLBK	Color or luminance of background
53275	DØ1B	PRIOR	Priority select
53276	DØ1C	VDELAY	Vertical delay
53277	DØ1D	GRACTL	Graphics control
53278	DØ1E	HITCLR	Clear collision flag
53279	DØ1F	CONSOL	Clear console switches

READ CTIA addresses

53248	DØØØ	MØPF	Missile Ø to playfield collision
53249	DØØ1	M1PF	Missile 1 to playfield collision
5325Ø	DØØ2	M2PF	Missile 2 to playfield collision

53251	DØØ3	M3PF	Missile 3 to playfield collision
53252	DØØ4	PØPF	Player Ø to playfield collision
53253	DØØ5	P1PF	Player 1 to playfield collision
53254	DØØ6	P2PF	Player 2 to playfield collision
53255	DØØ7	P3PF	Player 3 to playfield collision
53256	DØØ8	MØPL	Missile Ø to player collision
53257	DØØ9	M1PL	Missile 1 to player collision
53258	DØØA	M2Pl	Missile 2 to player collision
53259	DØØB	M3PL	Missile 3 to player collision
5326Ø	DØØC	PØPL	Player Ø to player collision
53261	DØØD	P1PL	Player 1 to player collision
53262	DØØE	P2Pl	Player 2 to player collision
53263	DØØF	P3Pl	Player 3 to player collision
53264	DØ1Ø	TRIGØ	Read trigger button Ø
53265	DØ11	TRIG1	Read trigger button 1
53266	DØ12	TRIG2	Read trigger button 2
53267	DØ13	TRIG3	Read trigger button 3

POKEY Chip (D2ØØ - D2ØF)

WRITE POKEY addresses

5376Ø	D2ØØ	AUDF1	Audio channel 1 frequency
53761	D2Ø1	AUDC1	Audio channel 1 control
53762	D2Ø2	AUDF2	Audio channel 2 frequency
53763	D2Ø3	AUDC2	Audio channel 2 control
53764	D2Ø4	AUDF3	Audio channel 3 frequency
53765	D2Ø5	AUDC3	Audio channel 3 control
53766	D2Ø6	AUDF4	Audio channel 4 frequency
53767	D2Ø7	AUDC4	Audio channel 4 control
53768	D2Ø8	AUDCTL	Audio control
53769	D2Ø9	STIMER	Start timers
5377Ø	D2ØA	SKRES	Reset status (SKSTAT)
53771	D2ØB	POTGO	Start potentiometer scan sequence
53772	D2ØD	SEROUT	Serial port output register
53773	D2ØE	IRQEN	IRQ interrupt enable
53774	D2ØF	SKCTL	Serial port 4 key control

READ POKEY addresses

5376Ø	D2ØØ	POTØ	Read potentiometer Ø
53761	D2Ø1	POT1	Read potentiometer 1
53762	D2Ø2	POT2	Read potentiometer 2
53763	D2Ø3	POT3	Read potentiometer 3
53764	D2Ø4	POT4	Read potentiometer 4
53765	D2Ø5	POT5	Read potentiometer 5
53766	D2Ø6	POT6	Read potentiometer 6
53767	D2Ø7	POT7	Read potentiometer 7
53768	D2Ø8	ALLPOT	Read 8 line pot. port state
53769	D2Ø9	RANDOM	Random number generator
53772	D2ØD	SERIN	Serial port input register
53773	D2ØE	IRQST	IRQ interrupt status register
53774	D2ØF	SKSTAT	Serial port 4 key status register

PIA Chip (D3ØØ - D3ØF)

54Ø16	D3ØØ	PORTA	Jack Ø & 1
54Ø17	D3Ø1	PORTB	Jack 2 & 3
54Ø18	D3Ø2	PACTL	Port A control

54Ø19	D3Ø3	PBCTL Port B control

ANTIC Chip (D4ØØ - D4ØF)

WRITE ANTIC addresses

54272	D4ØØ	DMACTL Direct memory acess control register
54273	D4Ø1	CHACTL Character control register
54274	D4Ø2	DLISTL Display list pointer (low byte)
54275	D4Ø3	DLISTH Display list pointer (high byte)
54276	D4Ø4	HSCROLL Horizontal scroll register
54277	D4Ø5	VSCROLL Vertical scroll register
54279	D4Ø7	PMBASE Player-missile base address register
54281	D4Ø9	CHBASE Character base address register
54282	D4ØA	WSYNC Wait for horizontal blank sync.
54286	D4ØE	NMIEN Non maskable interupt enable
54287	D4ØF	NMIRES Reset NMI status

READ ANTIC Addresses

54283	D4ØB	VCOUNT Vertical line counter
54284	D4ØC	PENH Horizontal light pen register
54285	D4ØD	PENV Vertical light pen register
54287	D4ØF	NMIST Non maskable interupt status register

Floating point routines
(use FRØ (ØØD4) FR1 (ØØEØ)
and ØØD4 - ØØFF, Ø57E - Ø5FF)

55526	D8E6 -	FASC	floating point to ASCII conversion
55722	D9AA -	IFP	integer to floating point conversion
55762	D9D2 -	FPI	floating point to integer conversion
58876	DA44 -	ZFRØ	zero FRØ (ØØD4)
58878	DA46 -	ZFR1	zero FR1 (ØØEØ)
559Ø4	DA6Ø -	FSUB	floating point subtraction
5591Ø	DA66 -	FADD	floating point addition
56Ø27	DADB -	FMULT	floating point multiplication
561Ø4	DB28 -	FDIV	floating point division
5664Ø	DD4Ø -	PLYEVL	floating point polynomial evaluation
56713	DD89 -	FLDØR	load floating point number to FRØ
56717	DD8D -	FLDØP	pointer to floating point number
56728	DD98 -	FLD1R	load floating point number to FR1
56732	DD9C -	FLD1P	pointer to floating point number
56743	DDA7 -	FSTØR	store floating point number from FRØ
56747	DDAB -	FSTØP	pointer to floating point number
56758	DDB6 -	FMOVE	move number from FRØ to FR1
56768	DDCØ -	EXP	floating point exponentiation (e)
56776	DDC8 -	EXP1Ø	floating point exponentiation (1Ø)
56781	DECD -	LOG	floating point natural logarithm
56785	DED1 -	LOG1Ø	floating point logarithm to base 1Ø

E4ØØ Screen editor handler base address

69184	E4ØØ	OPEN
69186	E4Ø2	CLOSE
69188	E4Ø4	GET
6919Ø	E4Ø6	PUT
69192	E4Ø8	STATUS
69194	E4ØA	JMP Power on

	E410		Display handler base address
69200		E410	OPEN
69202		E412	CLOSE
69204		E414	GET
69206		E416	PUT
69208		E418	STATUS
69210		E41A	JMP Power on
	E420		Keyboard handler base address
69216		E410	OPEN
69218		E412	CLOSE
69220		E414	GET
69222		E416	PUT
69224		E418	STATUS
69226		E41A	JMP Power on
	E430		Printer handler base address
69248		E430	OPEN
69250		E432	CLOSE
69252		E434	GET
69254		E436	PUT
69256		E438	STATUS
69258		E43A	JMP Power on
	E440		Cassette handler base address
69264		E440	OPEN
69266		E442	CLOSE
69268		E444	GET
69270		E446	PUT
69272		E448	STATUS
69274		E44A	JMP Power on
	E450		Disk handler vector addresses
69280		E450	JMP Disk initialization
69283		E453	DSKINV JMP Disk interface
69286		E456	JMP CIO
69289		E459	JMP SIO
69292		E45C	JMP SETVBL
69295		E45F	JMP SYSVBL
69298		E462	JMP XITCBL
69301		E465	JMP SIOINT
69304		E468	JMP SENDER
69307		E46B	INTINT
69310		E46E	CIOINT
69313		E471	Blackboard vector
69316		E474	Warm start vector (RESET)
69319		E477	Cold start vector (Power on)
69322		E47A	Read cassette block

$15.95

THE CREATIVE ATARI

Edited by David Small, Sandy Small and George Blank

Here's an invaluable reference guide for the non-expert Atari user who has only a limited knowledge of Basic and simple programming.

Containing up-dated and revised articles, reviews, and tutorials which previously appeared in Creative Computing magazine (along with some new material), The Creative Atari includes the following chapters:

- A "how-to" on Atari graphics
- Hardware and software reviews
- A discussion on the Atari disk drive
- Ready-to-run program listings

There's also an appendix with useful programming information.

And lots MORE!!

CREATIVE COMPUTING PRESS

ISBN 0-916688-34-8